НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Теплоенергетичний факультет
Кафедра автоматизації проектування енергетичних процесів і систем

До захисту допущено:
Завідувач кафедри
_____ Олександр Коваль
«___»_____2020 р.

Дипломна робота

на здобуття ступеня бакалавра

за освітньо-професійною програмою «Програмне забезпечення розподілених систем»

спеціальності 121 «Інженерія програмного забезпечення»

на тему: «Реалізація процедур для побудови каркасних стерео зображень з урахуванням багаторакурсного спостереження»

Виконав:
студент IV курсу, групи ТВ-61
Голотюк Павло Олександрович _____

Керівник:
д.т.н., ст.н.с., проф. каф. АПЕПС ТЕФ, Груць Ю.М. _____

Рецензент:
д. т. н., проф. СТАСЮК О. І. _____

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Київ – 2020 року

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет теплоенергетичний

Кафедра автоматизації проектування енергетичних процесів і систем

Рівень вищої освіти перший рівень

Напрям підготовки: 121 Інженерія програмного забезпечення

Спеціалізація: Програмне забезпечення розподілених систем

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ Олександр Коваль
(підпис)
”___”_____2020р.

ЗАВДАННЯ
на дипломну роботу студенту

Голотюку Павлу Олександровичу
(прізвище, ім’я, по батькові)

1. Тема роботи реалізація процедур для побудови каркасних стерео зображень з урахуванням багаторакурсного спостереження
керівник роботи проф. каф. АПЕПС, д.т.н., ст.н.с. Груць Ю.М.
(прізвище, ім’я, по батькові науковий ступінь, вчене звання)

затверджена наказом вищого навчального закладу від ”25” травня 2020р. **№ 1168-с**

2. Строк подання студентом роботи _____
3. Вихідні дані до роботи мова програмування C++, операційна система Microsoft Windows 10

4.Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити) Дослідити матеріали, надані керівником дипломної роботи, та інші матеріали, що є необхідними для виконання дипломної роботи. Розробити процедури для побудови каркасних стереозображень. Реалізувати багаторакурсне спостереження. Розробити програмний продукт для реалізації основних тривимірних процедур взаємодії між реальним і віртуальним світом на основі теорії стереооператорів для точково-скелетних 3D-зображень, з можливістю введення зворотного зв'язку по стереоракурсу.

5. Перелік ілюстративного матеріалу
_Каркасні стереозображення, постановка задачі, процес роботи, стереоперетворення, перетворення об’єктів, використані технології, графічний інтерфейс, інструменти малювання, трансформація об’єктів, робота з файловою системою, багаторакурсність, демонстрація роботи, висновки.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання ”11” жовтня 2019 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітки
1.	Затвердження теми роботи	08.01.2020	
2.	Вивчення та аналіз задачі	03.09.2019-04.12.2019	
3.	Розробка архітектури та загальної структури системи	04.12.2019-21.01.2020	
4.	Розробка структур окремих підсистем	21.01.2020-10.03.2020	
5.	Програмна реалізація системи	10.03.2020-30.05.2020	
6.	Оформлення пояснювальної записки	30.05.2020-06.06.2020	
7.	Захист програмного продукту	10.06.2020	
8.	Передзахист	10.06.2020	
9.	Захист	16.06.2020	

Студент _____ Голотюк П.О.
(підпис) (прізвище та ініціали,)

Керівник роботи _____ Груць Ю.М.
(підпис) (прізвище та ініціали,)

АНОТАЦІЯ

Робота займає 51 сторінок, містить 40 рисунків та 16 посилань. Метою роботи є розробка програмного продукту для реалізації основних тривимірних процедур взаємодії між реальним і віртуальним світом на основі теорії стереооператорів для точково-скелетних 3D-зображень, з можливістю введення зворотного зв'язку по стереоракурсу. В ході роботи було розроблена тестові проекти на мовах програмування C# та C++, досліджено принципи роботи стерео зображення та перетворень об’єктів, таких як поворот та переміщення у світових та локальних координатах. Розроблено алгоритми конвертації світових координат у локальні та екранні, з урахуванням стереовиводу. Розроблено алгоритм для повороту об’єктів за допомогою кватерніонів. Отримано досвід роботи з бібліотеками OpenGL, OpenCV, ImGui. Розроблено програмний продукт, що реалізує базовий функціонал для програми свого класу.

ABSTRACT

This work comprises 51 pages, contains 40 images and 16 references. The purpose of this work is to develop a software product for the implementation of basic three-dimensional procedures of interaction between the real and virtual world based on the theory of stereo operators for wire-frame 3D-images, with the possibility of stereo feedback. In course of this work were developed separate test projects in C# and C++ programming languages. Principles of stereo imaging and object transformation were studied, such as rotation and translation in world and local coordinates. An algorithm for converting world coordiantes to local and screen coordinates taking stereo output into account was developed. An algorithm for quaternion object rotation was created. Received working experience with libraries OpenGL, OpenCV, ImGui. Developed a software that implements basic functionality for its class of programs.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

- Група перетворення – повний набір перетворень об’єкта;
- Каскадне перетворення – послідовне застосування груп перетворення об’єкта до цього об’єкта;
- Пряме каскадне перетворення – каскадне перетворення за напрямом знизу угору (від листового об’єкта до кореневого);
- Зворотнє каскадне перетворення – пряме каскадне перетворення у якому групи перетворення застосовуються у зворотньому порядку та елементи групи є оберненими;
- Кадр – ітерація головного циклу програми;
- Json – формат даних, побудований по принципу ключ-значення, де значеннння може бути примітивним типом, або іншим Json об’єктом;

ВСТУП

Каркасні графічні 3D зображення є важливим класом в спектрі інших завдань просторової візуалізації. На екрані монітора каркасні комп'ютерні тривимірні зображення представлені набором точок і / або векторів. Каркасні зображення широко використовуються для проектування різноманітних систем та роботи з тривимірною графікою загалом. Однак обмеженням типового каркасного зображення є його «плоскість», що примушує користувача переміщувати точку спостереження для того, щоб спостерігати третю вісь координат.

Стерео зображення дозволяє відчувати «глибину» зображення завдяки використання переваг бінокулярної структури людського бачення. Робота з каркасними зображеннями, що не представляють з себе впізнаваної форми, такими як водяні потоки у річці, магнітні поля, атомна структура органічних молекул тощо, є проблемною областю, що вимагає побудову додаткових перерізів, діаграм чи інших засобів візуалізації через те, що традиційні двовимірні зображення не дають можливості швидко визначити зону інтересу. На даному етапі тільки стереоскопічні 3D-системи дають можливість адекватно вирішити таку задачу.

Однак недоліком стерео-графіки є її залежність від ракурсу спостереження. При невідповідності положення спостерігача у просторі та розрахованого положення, на базі якого було побудовано зображення спостерігач може відчувати дискомфорт. Реалізація багаторакурсного спостереження вирішує цю проблему та додає новий рівень взаємодії, коли користувач може переміщувати точку спостереження для того щоб заглянути за об’єкт.

1. ЗАДАЧА РЕАЛІЗАЦІЇ ПРОЦЕДУР ДЛЯ ПОБУДОВИ КАРКАСНИХ СТЕРЕО ЗОБРАЖЕНЬ З УРАХУВАННЯМ БАГАТОРАКУРСНОГО СПОСТЕРЕЖЕННЯ

Метою роботи є розробка програмного продукту для реалізації основних тривимірних процедур взаємодії між реальним і віртуальним світом на основі теорії стереооператорів для точково-скелетних 3D-зображень, з можливістю введення зворотного зв'язку по стереоракурсу.

Функції, які треба реалізувати:

- Конвертація між тривимірними координатами об’єкта, та стереокоординатами монітора;
- Щуп (тривимірний курсор);
- Малювання кривих та ламаних;
- Малювання кривими та ламаними;
- Переміщення об’єктів;
- Поворот об’єктів;
- Масштабування об’єктів;
- Модифікація стереокоординат за допомогою відстеження позиції користувача;
- Завантаження\збереження об’єктів у файл.

Для виконання даної задачі необхідно:

1. Виконати огляд існуючих рішень, що реалізують роботу із стереоскопічним зображенням. Проаналізувати реалізацію роботи із стереозображенням обраних рішень, їх переваги та недоліки;

2. Дослідити основи принципів роботи з стереозображеннями та багаторакурснісню, що необхідно реалізувати у програмному продукті. Дослідити основи роботи з об’єктами у тривимірному просторі;
3. Обрати цільову платформу для реалізації програмного продукту. Проаналізувати доцільність та можливість реалізації проекту на мовах C# та C++. Створити тестові проекти на цих мовах для дослідження основ роботи з графікою та графічним інтерфейсом у різних мовах. Визначитися зі сторонніми бібліотеками для реалізації проекту на обраній платформі та мові програмування;
4. Розробити архітектуру майбутнього продукту. Розробити інфраструктуру для взаємодії зі сторонніми бібліотеками та обраною операційною системою;
5. Реалізувати графічний інтерфейс програми, стереовивід сцени із урахуванням багаторакурсності та обраний функціонал;

2. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

2.1. Загальні відомості

Наразі існує багато програмних рішень для роботи з тривимірною графікою. Багато з них також реалізує роботу із стерео-зображеннями (Blender, Cinema4D, 3DSMax та ін.). На рисунку 2.2 зображено вікно програми Blender з налаштованою стерео-камерою. Однак ці програми не реалізують багаторакурсне спостереження. Більшість програм, що реалізовували цю задачу є вузько спеціалізованими, та не знаходяться у відкритому доступі. Прикладом таких програмних рішень є програмне забезпечення для ультразвукових апаратів (УЗІ), магнітно-резонансних томографів (Рисунок 2.2), рентгенові сканери багажу в аеропортах, тривимірне сканування предметів (Рисунок 2.1), місцевості тощо [1].

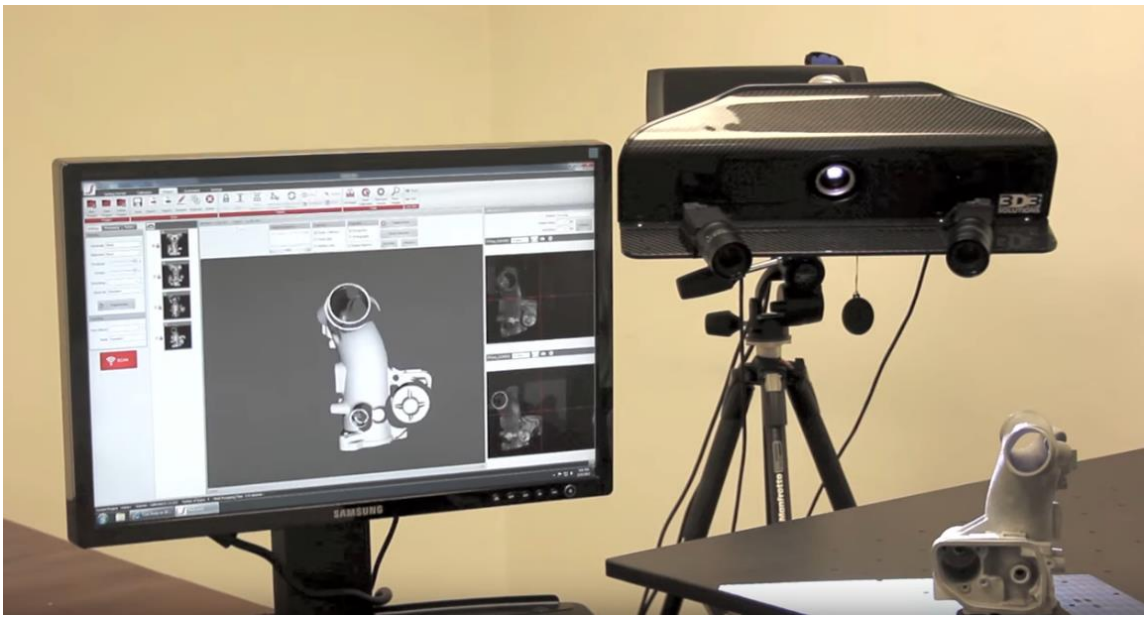


Рисунок 2.1 – програма для сканування тривимірних об’єктів

Однак більшість рішень лише використовують багаторакурсність як інструмент для отримання інформації про просторові координати точок на об’єкті для подальшого створення його тривимірної моделі і не пропонують стереовиводу на монітор. Так, запропонований Z. Stankovic та B. D. Allen у їх статті про чотиривимірну

візуалізацію МРТ [2] метод зображає потік крові у судинах за допомогою побудови чотиривимірної моделі з набору МРТ знімків, отриманих з різних ракурсів, що можна бачити на рисунку 2.2. Однак, з підвищенням складності моделі, криві, що символізують порцію потоку починають зливатися і модель доводиться повертати задля кращого розуміння усієї сцени. Цей недолік обмежує складність можливих побудованих моделей.

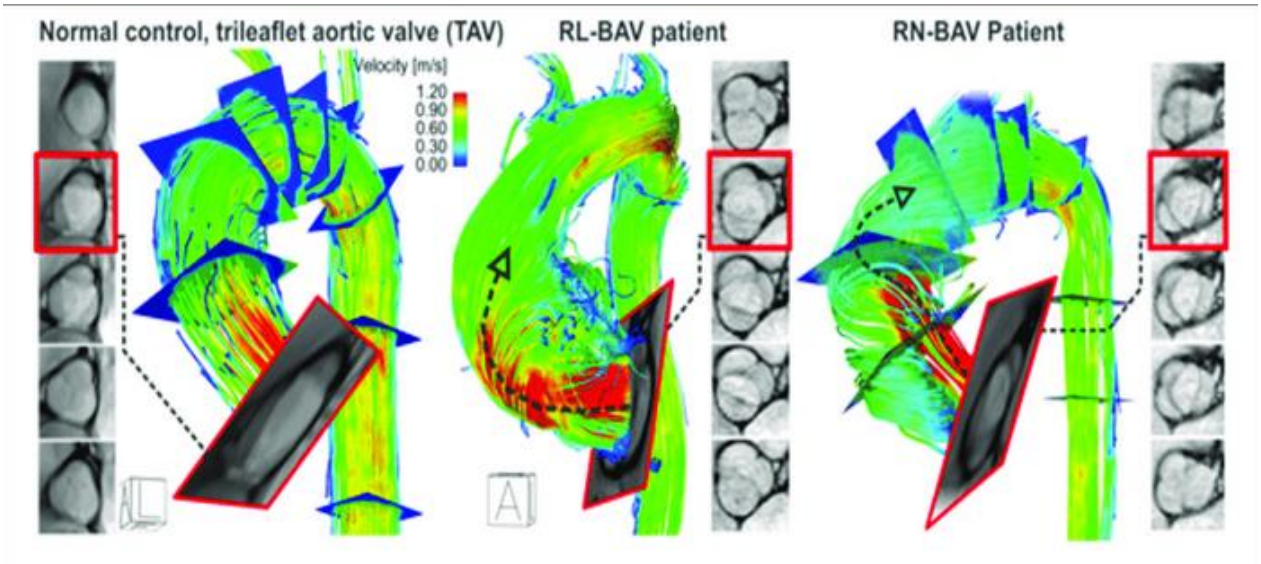


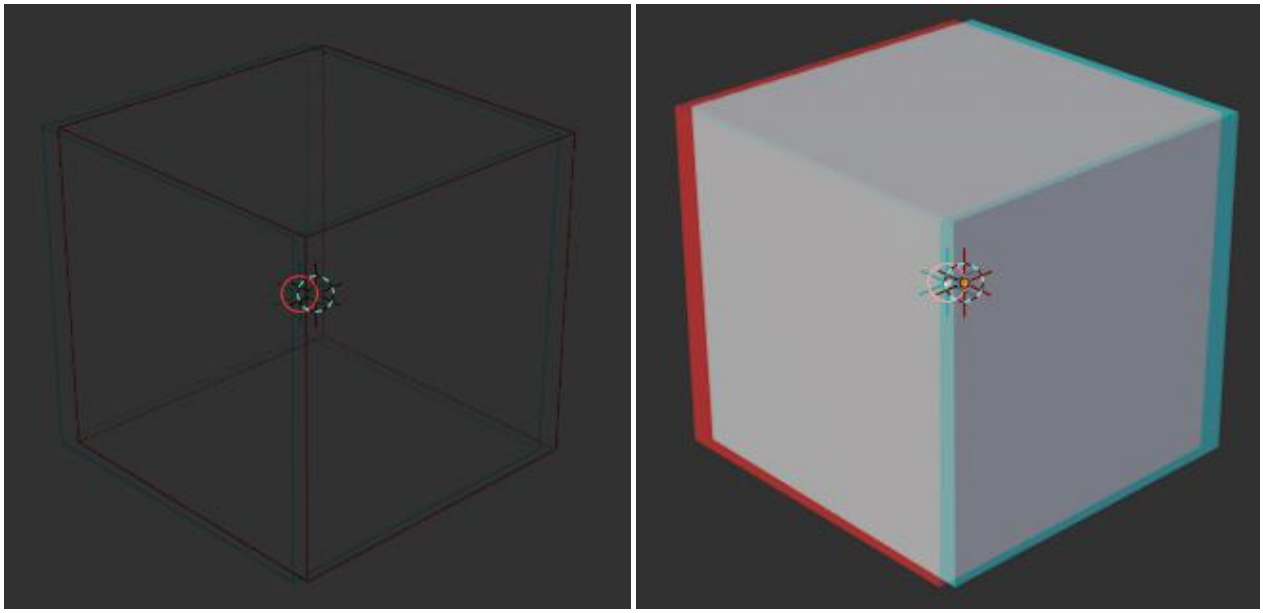
Рисунок 2.2 – візуалізація току крові у різних пацієнтів

2.2. Blender

Blender є програмою для моделювання тривимірних сцен загального призначення, є безкоштовною але є на перший погляд не досить інтуїтивною у використанні. Для того, щоб увімкнути стереовивід необхідно перейти у робочий простір (workspace) «Rendering», далі на правій панелі властивостей обрати вкладинку «вивід» (output) та встановити прапорець на опцію «Стереоскопія» (Stereoscopy).

Тепер, якщо перейти у будь-який робочий простір та обрати вид з камери (View -> Cameras -> Active camera), то можна бачити що сцена відмальовується у стереорежимі (рисунок 2.3). А на панелі властивостей камери можна бачити налаштування стереоскопії, такі як відстань до глядача до екрану, чи відстань між

очима спостерігача (рисунок 2.4). Легко бачити, що скелетне зображення у стереовиводі є темним та не зручним у роботі та демонстрації.



(а) (б)

Рисунок 2.3 – Стерео-вивід скелетного (б) та заповненого (а) зображень у програмі Blender

Варто зазначити, що виділені об’єкти стають яскравими, але кольори для виводу часто інвертуються, при цьому коли усі інші об’єкти залишаються темними, що тільки заважає під час роботи у стереоокулярах.

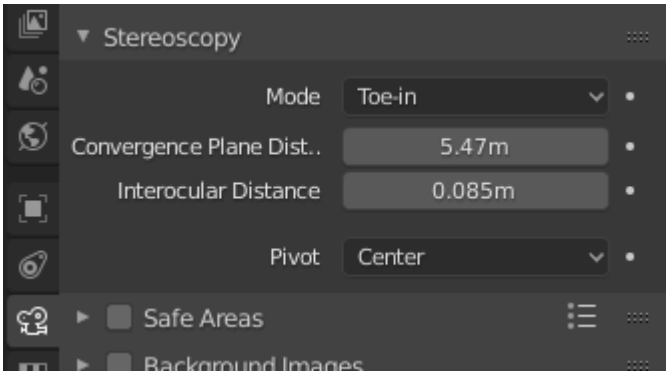


Рисунок 2.4 – властивості стереоскопії камери

У свою чергу, Blender реалізує досить детальну роботу з малювання прямих та кривих різними методами, такими як криві Безьє чи Nurbs. Проте, не реалізовано виводу цих кривих у стереорежимі. Вони не вважаються фізичними об’єктами, а відповідно і не відображаються у виді з камери.

3. ПОБУДОВА СТЕРЕОЗОБРАЖЕННЯ ІЗ БАГАТОРАКУРСНІСНЮ

3.1. Каркасні стерео зображення

Каркасні графічні 3D зображення є важливим класом в спектрі інших завдань просторової візуалізації. На екрані дісплея каркасні комп'ютерні тривимірні зображення представлені набором точок і / або векторів.

Нижче наведено перелік задач, вирішення яких призводить до візуалізації тривимірних зображень точково-скелетного виду, а саме:

- Задачі побудови траєкторій. Візуалізація результатів натурних випробувань, напівнатурного або комп'ютерного моделювання поведінки рухомих об'єктів у різноманітних середовищах (підводне середовище [3], космос, атмосфера); завдання локації;
- Польові задачі. Силові лінії поля (теплового, електромагнітного [4] тощо);
- Хіміко-фізичні задачі. Тривимірні моделі органічних молекул, кристалічних структур вірусів, кристалічних решіток, ядер атомів;
- Контурні задачі. Завдання, що потребують візуалізації контурів перетину тривимірних об'єктів площинами на різних рівнях по третій координаті, такі як технічна та медична томографії[2], картографія, підготовка контурів для волюметричних систем тощо;
- Задачі розподілу. Розташування в просторі об'єктів відносно невеликої величини в порівняно з займаним обсягом (моделі зоряного неба; домішок у розчинах і розплавах; карти підводних мінних полів тощо);
- Задачі навчання. Креслення, нарисна геометрія, стереометрія;
- Задачі дизайну та проектування (як основа для напівтонової 3D графіки);
- Задачі натурно-комп'ютерного графічного стереомоделювання та задачі каркасного копіювання з натурних стереозображень.

Принцип побудови стереоскопічних проекцій для графічних тривимірних зображень добре відомий [5].

В ІПМЕ ім. Г.Є. Пухова НАН України запропоновано і розроблено новий операторний метод стерео перетворень [6,7,8], суть якого полягає в тому, що знайдено формальний математичний апарат, який встановлює взаємно-однозначну відповідність між тривимірними координатами довільної точки шуканого об'єкта, заданими в світовій системі координат, і стереокоординатами цієї точки, заданими в екранній системі координат. Перехід з просторової ділянки в стереоділянку виконується за допомогою, так званого, прямого оператора стереоперетворення; $S\{\vec{V}\} \Rightarrow \vec{s}$; зворотний перехід здійснюється за допомогою зворотного оператора стереоперетворення $S^{-1}\{\vec{s}\} \Rightarrow \vec{V}$ (рис.3.1).

Такий підхід дав можливість вирішити проблему пов'язаних (кореспондуючих) точок для графічних комп'ютерних стереозображень.

Отримано матричні залежності, що відображають дане перетворення для випадку, коли спостерігач знаходиться в точці постійного стереоракурсу.

Прямий і зворотний стереооператори постійного ракурсу реалізуються за допомогою таких залежностей:

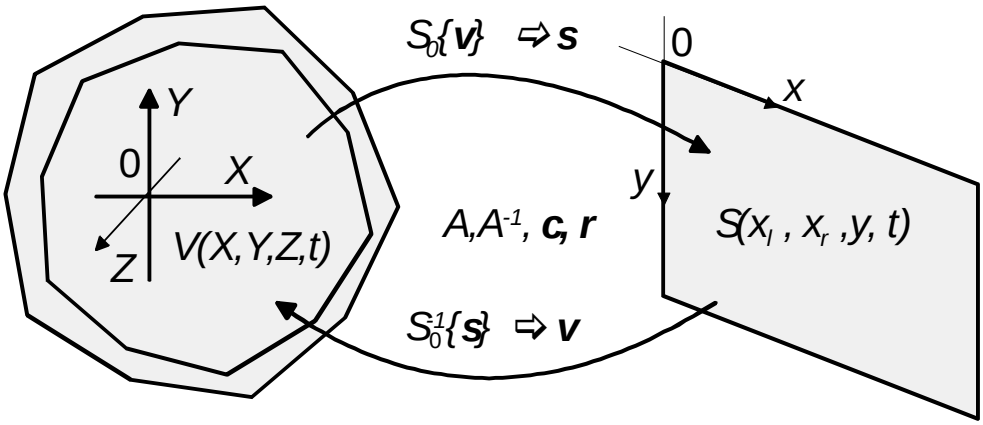


Рисунок 3.1 – Операторний метод стереоперетворення

$$\vec{s} = A(\vec{V} + \vec{c})/(z_0 - Z - \dot{z}) \,, \qquad (3.1)$$

$$\vec{V} = (z_0 - Z - \dot{z})/A^{-1}\vec{s} - \vec{c} \,, \qquad (3.2)$$

де $2a$ – стереобазис, A, A^{-1} – пряма та зворотна квадратні матриці,

$$A = \begin{bmatrix} z_0 & 0 & a - x_0 \\ z_0 & 0 & -a - x_0 \\ 0 & -z_0 & -y_0 \end{bmatrix},\ A^{-1} = \frac{1}{2az_0} \begin{bmatrix} (a + x_0) & (a - x_0) & 0 \\ -y_0 & y_0 & -2a \\ z_0 & -z_0 & 0 \end{bmatrix},$$

$$\vec{V} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix},\ \vec{c} = \begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{bmatrix},\ \vec{r} = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix},\ \vec{s} = \begin{bmatrix} x_l \\ x_r \\ y \end{bmatrix},$$

$\vec{V}$ – вектор тривимірних координат довільної точки об'єкта,

$\vec{s}$ – вектор стереокоординат шуканої точки,

$\vec{c}$ – вектор зміщення початку координат світової і екранної систем,

$\vec{r}$ – вектор постійного стереоракурсу.

3.2. Багаторакурсне спостереження

Розпізнавання об’єктів користуючись каскадною класифікацією за ознаками Хаара є ефективним методом розпізнавання об'єктів запропонованим Paul Viola та Michael Jones в їх статті про методи швидкого розпізнавання образів [9]. Був використаний підхід машинного навчання, у якому каскадна функція тренована з великої кількості позитивних на негативних зображень. Що далі використовується для розпізнавання об’єктів у інших зображеннях.

Алгоритм розпізнавання обличчя, що є засобом реалізації багаторакурсного спостереження запропонований в даній роботі має бути натренований великою кількістю позитивних зображень (зображення з обличчами), та негативних (зображення без облич). Далі необхідно виділити риси обличчя. Для цього використовуються риси Хаару, зображені на рисунку 3.2, що нагадують ядра згортки (convolutional kernel) [10]. Кожна риса це єдине значення отримане відніманням пікселів під білим прямокутником від суми пікселів під чорним прямокутником.

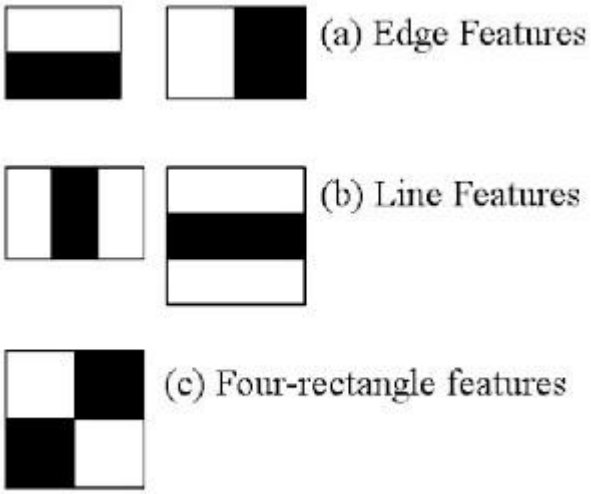


Рисунок 3.2 – Риси Хаару

Далі необхідно обчислити усі можливі варіації розмірів та положень кожної риси, що вимагає величезних ресурсів. Для вирішення цієї проблеми було запропоновано інтегрування зображень [9]. Завдяки цьому підходу обчислення для кожного піксела перетворюється на обчислення усього чотирьох пікселів незалежно від розміру зображення. Це значно прискорює роботу. Однак серед знайдених рис більшість не є шуканими. Одні й ті самі риси можуть бути знайдені у багатьох місцях зображення.

Для виокремлення бажаних рис використовується машинне навчання. Для цього, відбувається тренування для кожної риси, що знаходить найкраще співвідношення за яким обличчя будуть класифікуватися на позитивні чи негативні. У результаті отримується великий набір рис, багато з яких помилкові чи неточні. Обираються риси з найменшою кількістю помилок та процес повторюється до тих пір поки не досягнуто бажаної точності розпізнавання рис.

Подальше пришвидшення алгоритму досягається за допомогою ідеї Каскадних класифікаторів. Це дозволяє розділити риси на різні етапи класифікаторів та застосовувати їх послідовно. Зображення поділяється на кілька вікон і відбувається пошук рис у кожному вікні відповідно до етапу класифікатора. Завдяки цьому області зображення, що не містять рис відповідного етапу відкидаються на ранніх етапах, що значно знижує кількість необхідних подальших обчислень.

3.3. Адаптація алгоритму

Реалізація багаторакурсності результує у необхідності перераховувати ракурс спостереження кожного кадру що, за умови використання формули (3.1) призводить до потреби обчислювати оригінальні координати стереопари за формулою (3.2), що є ресурсовитратним процесом. Для спрощення обчислень запропоновано перейти від «об'єкто-центричної» до «камеро-центричної» моделі. Тоді формула для отримання екранних координат виглядатиме наступним чином:

$$\vec{s} = \begin{bmatrix} (x_0'X - (a - x_0)Z)/d \\ (x_0'X - (a + x_0)Z)/d \\ z_0'(-Y) + y_0'Z/d \end{bmatrix}, \quad \vec{r}' = \begin{bmatrix} x_0{}' \\ y_0{}' \\ z_0{}' \end{bmatrix} = \vec{r} + \overrightarrow{c_u}, \qquad d = z_0' - Z,$$

де $\vec{r}'$ – трансформований вектор постійного стереоракурсу,

$\vec{r}$ – попередньо визначений стереоракурс або початкові координати камери,

$\overrightarrow{c_u}$ – координати спостерігача перед монітором або модифікатор координат камери.

Завдяки такому підходу усі операції над об'єктами відбуваються у «традиційному» тривимірному просторі, що зменшує кількість необхідних обчислень для різноманітних інструментів, реалізованих в даній роботі, зменшує кількість використованої пам'яті та підвищує простоту та надійність початкового коду програми.

4. ПЕРЕТВОРЕННЯ ОБ’ЄКТІВ

4.1 Каскадні перетворення

Для виконання поставленої задачі було обрано реалізацію каскадної, або акумуляційної системи перетворень. Ця система дозволяє реалізувати локальну та глобальну (світову) систем координат для об’єктів сцени, що є основою для будь-якого тривимірного редактора.

Каскадна система перетворень представляє з себе набір функцій, що конвертують групи перетворень об’єктів з локальних у світові координати, чи навпаки. Наприклад, для того, щоб отримати координати об’єкта для подальшого його малювання на екрані необхідно перевести його у глобальні координати враховуючи усі його властивості, такі як поворот, масштаб чи позиція.

Базовий алгоритм такої функції зображений на рисунку 4.1. Тут, точка, яка передається у функцію за посиланням модифікується усіма перетвореннями послідовно, таким чином піднімаючись угору по ієрархії об’єктів. Процес

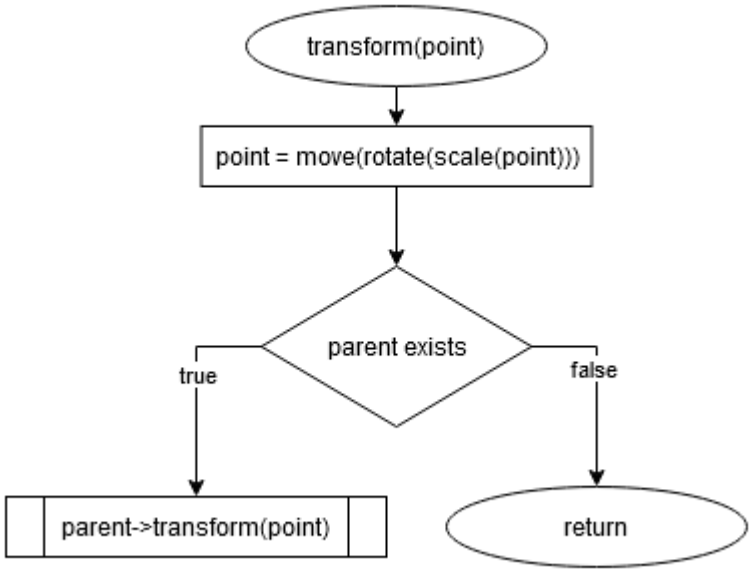


Рисунок 4.1 – Каскадне перетворення точки об’єкта

повторюється, поки не буде досягнуто кореневого об’єкта, який є батьківським для усіх об’єктів сцени. Таку трансформацію можна назвати прямою каскадною трансформацією. Іншими словами світові координати у сцені є ні чим іншим як локальними координатами відносно кореневого об’єкта.

Разом з тим, для конвертації глобальних у локальні координати необхідна функція зворотньої (оберненої) каскадної трансформації. Усі трансформації замінені на обернені та застосовуються у зворотному порядку.

4.1 Поворот

Для реалізації каскадної системи перетворень необхідно зберігати також і поворот об’єкта. Для представлення повороту об’єкта у просторі було обрано кватерніон, що є комплексним числом четвертого порядку. Важливою властивістю векторного добутку кватерніонів є його асоціативність, що є дуже корисним для конвертування об’єктів між локальними та світовими координатами. Поворот вектора кватерніоном відбувається за наступною формулою:

$$v' = q \otimes v \otimes q^{-1}, \quad (4.1)$$

Де q – кватерніон повороту,

q^{-1} – обернений кватерніон повороту,

v – початковий вектор,

v' – результуючиій вектор,

$\otimes$ – операція векторного добутку кватерніонів.

При цьому вектор v у рівнянні 4.1 береться як чистий кватерніон, тобто кватерніон, дійсна частина якого дорівнює нулю [10].

Однак така операція є досить затратною, беручи до уваги, що її доводиться обчислювати багато разів. Для спрощення обчислень можна скористатися тим припущенням, що вектор v є чистим кватерніоном ($v_w = 0$) та переписати рівняння наступним чином:

$$v' = q \odot v = 2\,\overrightarrow{q_v} \times (\overrightarrow{q_v} \times v + q_w v) + v, \quad (4.2)$$

Де × – векторний добуток векторів,

⊙ – операція повороту вектора.

Таким чином, формули отримання та встановлення світового повороту об’єкта відповідно виглядатиме наступним чином:

$$q^w = q^{pw} \otimes q^l, \qquad (4.3)$$

$$q^{l\prime} = (q^{pw})^{-1} \otimes q^{w\prime}, \qquad (4.4)$$

Де q^{pw} – поточний світовий поворот батьківського об’єкта,

q^w – поточний світовий поворот об’єкта,

q^l – поточний локальний поворот об’єкта,

$q^{l\prime}$ – новий локальний поворот об’єкта,

$q^{w\prime}$ – новий світовий поворот об’єкта.

Однією з цікавих властивостей векторного добутку кватерніонів є відносність повороту множників відносно один одного. Це означає, що для повороту об’єкта у світових координатах необхідно помножити новий поворот на поточний, а у локальних – поточний на новий:

$$q^{w\prime} = q^{wd} \otimes q^w, \qquad (4.5)$$

$$q^{l\prime} = q^l \otimes q^{ld}, \qquad (4.6)$$

Де q^l – є поточним локальним поворотом об’єкта,

q^w – є поточним світовим поворотом об’єкта обчислений за формулою 4.3,

$q^{l\prime}$ та $q^{w\prime}$– є новими локальним та світовим поворотами об’єкта відповідно,

q^{ld} та q^{wd} є поворотами на які об’єкта буде повернуто у локальних та світових координатах відповідно.

Варто зауважити, що $q^{ld} = q^{wd}$ і що єдина різниця між кінцевим результатом їх множення з іншими поворотами є їх позиція у виразі. Повороти q^{ld} , q^{wd} представляють поворот точки навколо осі, і обчислюються наступним чином:

$$q^{ld} = q^{wd} = \begin{bmatrix} cos\,(\theta/2) \\ \overrightarrow{v_x} \\ \overrightarrow{v_y} \\ \overrightarrow{v_z} \end{bmatrix}, \qquad (4.7)$$

Де θ – кут повороту,

$\vec{v}$ – вісь повороту помножена на синус половини половини кута повороту. Причому напрям вектора співпадає з напрямком однієї з вісей координат. Тобто:

$$\vec{v} = \begin{bmatrix} \sin(\theta/2) \\ 0 \\ 0 \end{bmatrix} \text{ або } \vec{v} = \begin{bmatrix} 0 \\ \sin(\theta/2) \\ 0 \end{bmatrix} \text{ або } \vec{v} = \begin{bmatrix} 0 \\ 0 \\ \sin(\theta/2) \end{bmatrix}$$

4.1 Результуючі перетворення

Отже, кінцеві формули для прямої та зворотньої груп перетворень виглядають наступним чином:

$$\vec{v}^w = q^l \odot (\vec{v}^l s^l) + p^l \,, \qquad (4.8)$$

$$\vec{v}^l = q^{l^{-1}} \odot (\vec{v}^w - p^l)/s^l, \qquad (4.9)$$

Де $\vec{v}^w$ – точка у глобальних координатах,

$\vec{v}^l$ – точка у локальних координатах об’єкта,

q^l, s^l, p^l – локальні поворот, масштаб та позиція об’єкта.

Для виконання каскадних перетворень необхідно обчислити такі групи перетворень для кожного об’єкта у ланцюгу ієрархії об’єктів сцени.

5. ЗАСОБИ РОЗРОБКИ

Для розробки було запропоновано мови програмування C# та C++. Було створено прототипи на обох та протестовано продуктивність. C# переміг у кількості готового функціоналу з «коробки» надаючи готовий графічний інтерфейс у вигляді оболонки WPF (Windows Presentation Forms) та просте підключення OpenGL. Однак робота з OpenGL виявилася дуже не гнучкою – ті бібліотеки, що розроблені під C# і є, власне, адапторами C бібліотеки – не реалізують частини функціоналу та виконуються з незадовільною швидкістю. Підключення OpenGL як бібліотеку з некерованою пам’яттю до проекту на C# є дуже довгим та кропітким процесом, що вимагає додаткових досліджень для кращої взаємодії проектів з різними моделями керування пам’яті [11].

C++ є мовою нижчого рівня за C# за замовчуванням [11]. Під час розробки на C++ потрібно враховувати багато деталей, таких як ручне керування пам’яттю та уникання циклічної залежності [12]. Однак в наявності багато бібліотек для реалізації графічного інтерфейсу, так як стандартних рішень для його реалізації не пропонується. Досить висока продуктивність досягається без використання особливих знань про роботу компілятора, віртуальної машини тощо. Через свою наближеність до «заліза» багато бібліотек для роботи з графікою розробляються саме для C++ [13].

За показник продуктивності було обрано кількість відрізків, що програма може відмалювати, не опускаючи частоту кадрів нижче 30 за секунду. Для тестового проекту на C# цей показник сягнув 2500, для C++ - 6000.

Отже, за результатами тестів та аналізу переваг та недоліків реалізацій тестових проектів було обрано мову програмування C++.

Програма розроблена з використанням бібліотеки для роботи з графікою OpenGL 3, що є відкритим стандартом для роботи з графікою у сучасних системах [14]. Як лоадер OpenGL було запропоновано два варіанти: glue та glfw (gl3w). Було

обрано остнній через кращу роботу з сучасними платформами та менший розмір пакета, що включає лише базовий функціонал.

За графічну оболонку було обрано бібліотеку ImGui, що доволяє створювати вікна з використанням FBO (Frame Buffer Object), що використовується в даному проекті для реалізації вікна відмальовки заданої тривимірної сцени. Дана бібліотека спеціально розробляється для створення програм-редакторів та надає великий набір інструментів. ImGui знаходиться у відкритому доступі на ресурсі GitHub. Було обрано гілку Docker-window, так як в ній реалізовано зручний механізм кріплення вікон одне до одного, що широко використовується у різноманітних редакторах.

Для реалізації багаторакурсного спостереження було обрано бібліотеку OpenCV, що надає можливість роботи з комп’ютерним баченням використовуючи риси Хаару [15].

Використані бібліотеки розповсюджується за ліцензійною моделлю MIT, що є вигідним для даного проекту.

6. ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

6.1. Доменні типи (Domain types)

Доменними типами є типи, що представляють об’єкти, якими оперує бізнес логіка програми. У нашому випадку це сцена та об’єкти сцени, такі як: відрізок, ламана, камера, трикутник, тощо. Доменні типи організовані у ієрархічну структуру, що дозволяє користуватися перевагами поліморфізму. Операції каскадного перетворення згадані у розділі 4 реалізовані безпосередньо у доменних об'єктах.

6.2. Команди та події (Commands & Events)

Команди (Commands) є замкнутою архітектурою, що є замінником асинхронного виконання у C#. Вони не реалізують усі переваги асинхронного виконання, та більш адаптовані до виконання поставлених у даному проекті задач.

Команди накопичуються у черзі під час проходження ітерації головного циклу та запускаються на виконання у його кінці. Кожна команда має властивість готовності на виконання, що є своєрідною затримкою на виконання. Ця властивість гарантує, що команда не буде виконаною до тих пір, поки команда не стане готовою до виконання.

Одною з команд, де використовується властивість готовності є команда переміщення об’єктів сцени. Під час початку перетягування об’єктів сцени у вікні інспектора об’єктів створюється команда на переміщення об’єктів. Коли користувач відпускає мишку у місці він хоче перемістити об’єкт команда встановлюється як готова і в кінці ітерації головного циклу команда виконує переміщення об’єкту.

Команди мають бути розміщені у динамічній пам’яті та не бути видаленими користувачем. Команди видаляються автоматично після того, як були виконані.

Події зберігають набір функцій, що будуть виконані коли об’єкт-власник відправить команду на виконання. На подію може підписатися будь-який об’єкт, що має посилання на власника події. Події також є закритими об’єктами, що мають тільки декілька відкритих методів для додавання та видалення підписників на події. Додавання та видалення підписників відбувається за допомогою вище згаданих команд.

Варто зазначити, що події не виконуються разом з командами, у кінці головного циклу, а одразу після настання назначеної події. Тобто цілком можливо передати у подію функцію, що буде відписувати себе – буде створено команду на відписування що виконається відбудеться у кінці кадру, абсолютно безпечно. Але і підписання на подію відбувається за допомогою команди, тож не варто очікувати виконання підписаної функції у той же кадр у який вона була підписана.

На рисунку 6.1 зображено приклад використання команди на виконання функції у класі, що реалізує події для того, щоб поставити відписання функції від події у чергу.

```
template<typename...T>
class IEvent {
protected:
    std::map<size_t, std::function<void(T...)>> handlers;
public:
    size_t AddHandler(std::function<void(T...)> func) { ... }
    void RemoveHandler(size_t v) {
        (new FuncCommand())->func = [&, v = v] {
            handlers.erase(v);
        };
    }
};
```

Рисунок 6.1 – клас IEvent

6.3. Інструменти ти Вікна (Tools & Windows)

Інструменти поділяються на два види – інструменти створення та редагування. Перші користуються командами створення об’єктів для, власне, створення об’єктів сцени. Приймають посилання на сцену, та функцію ініціалізації об’єкта.

Інструменти редагування модифікують об’єкти сцени відповідно до функцій інструмента та вводу користувача. Відбувається підписання на ввід користувача та на зміни у конфігурації інструментів, таких як зміна режиму координат. Керування інструментами відбувається за допомогою клавіатури та миші, а також вікна інструмента, про що буде детальніше пояснено у розділі 7.

Інструменти зберігають оригінальні властивості прив’язаного до них об’єкта, щоб у разі помилки інструмент міг відмінити дію. Для початку роботи з інструментом його необхідно ініціалізувати, передавши параметри, що можуть відрізнятися для різних інструментів. Через це було вирішено реалізувати окремий клас, що називається ToolPool. Він надає статичні методи для створення, ініціалізації та отримання інструментів. Сам ToolPool також потребує ініціалізації, та приймає багато параметрів, щоб задовольнити потреби усіх інструментів. Приклад ініціалізації можна бачити на рисунку 6.2.

```
class ToolPool {
    static bool Init(PointPenEditingTool<StereoPolyLineT>* tool) {
        return
            tool->BindInput(*GetKeyBinding()) &&
            tool->BindCross(*GetCross());
    }

    static bool Init(ExtrusionEditingTool<StereoPolyLineT>* tool) {
        return
            tool->BindInput(*GetKeyBinding()) &&
            tool->BindCross(*GetCross()) &&
            tool->BindScene(*GetScene()) &&
            tool->BindDestination(&(*GetScene())->root);
    }
```

Рисунок 6.2 – фрагмент класу ToolPool

Вікна представляють з себе вікна у графічному середовищі ImGui. Вони можуть містити різноманітні графічні елементи управління, що можна бачити на рисунку 6.3.

Вікна містять три основні методи:

- Init, що викликається після того, як вікно було налаштовано. Цей метод перевіряє правильність налаштування та виконує остаточне налаштування вікна, таке як створення екземплярів внутрішніх об’єктів, створення команд, тощо;

- Design, що викликається під час рендеру графічного інтерфейсу та описує логіку взаємодії користувача з вікном;
- Exit, що викликається коли вікно закривається. Очищує пам’ять чи зупиняє інші процеси, що були пов’язані з вікном.

Вікна можуть взаємодіяти між собою через буфер об’єктів сцени, завдяки якому відбувається перетягування об’єктів сцени між вікнами.

```
bool DesignInternal() {
    ImGui::Text(GetName(target).c_str());

    if (ImGui::BeginDragDropTarget()) { ... }

    if (ImGui::Extensions::PushActive(*target != nullptr)) { ... }

    transformToolMode = (int)tool->GetMode();
    {
        if (ImGui::RadioButton("Transition", &transformToolMode, (int)TransformToolMode::Translate))
            tool->SetMode(TransformToolMode::Translate);
        if (ImGui::RadioButton("Scale", &transformToolMode, (int)TransformToolMode::Scale))
            tool->SetMode(TransformToolMode::Scale);
        if (ImGui::RadioButton("Rotate", &transformToolMode, (int)TransformToolMode::Rotate))
            tool->SetMode(TransformToolMode::Rotate);
    }

    if (transformToolMode == (int)TransformToolMode::Translate) {
        ImGui::Separator();
        DragVector(tool->transformPos, "X", "Y", "Z", "%.5f", 0.01);
    }
    else if (transformToolMode == (int)TransformToolMode::Scale) {
        ImGui::Separator();
        ImGui::DragFloat("scale", &tool->scale, 0.01, 0, 0, "%.2f");
    }
    else if (transformToolMode == (int)TransformToolMode::Rotate) {
        ImGui::Separator();
        DragVector(tool->angle, "X", "Y", "Z", 1);
    }
```

Рисунок 6.3 – налаштування інтерфейсу за допомогою бібліотеки ImGui

6.4. Рендерер (Renderer)

Renderer відповідає за взаємодію з OpenGL та малювання об’єктів сцени кожний кадр у FBO (Frame Buffer Object), який потім буде відображено у вікні програми у якості зображення ImGui. Вміє працювати тільки із стереовідрізками, тож для малювання складних об’єктів, як то многочленів потребує конвертації многочленів у відрізки. Це відбувається у методі GetLines, що реалізують усі об’єкти сцени. Цей метод повертає закешовані відрізки та виконує метод UpdateCache, що користується функцією каскадного перетворення (див. розділ 4) для конвертації об’єкта у світові координати. Перед початком рендерингу сцени викликається

фукнкція CustomRenderFunc. У зазначеній функції відбувається модифікація властивостей камери відповідно до результатів роботи детектора положення. Конвертація світових координат точок у екранні координати відбувається у об’єкті-камері. Камера також реалізує методи для збереження пропорцій отриманого зображення.

Внутрішньо, працює з трикутниками для малювання перетину правої та лівої частин об’єктів на екрані білим кольором. Дана функціональність реалізована за допомогою використання Stencil масок, як видно на рисунку 6.4.

```
void DrawObject(StereoCamera* camera, SceneObject* o) {
    for (auto l : o->GetLines()) {
        // ...
        glStencilMask(0x1);
        glStencilFunc(GL_ALWAYS, 0x1, 0xFF);
        glStencilOp(GL_KEEP, GL_REPLACE, GL_REPLACE);
        DrawLineLeft(camera->GetLeft(l));

        glStencilMask(0x2);
        glStencilFunc(GL_ALWAYS, 0x2, 0xFF);
        glStencilOp(GL_KEEP, GL_REPLACE, GL_REPLACE);
        DrawLineRight(camera->GetRight(l));
    }
}
```

Рисунок 6.4 – метод для рендерингу об’єктів

6.5. Графічний інтерфейс та головний цикл (GUI & main loop)

Клас GUI містить головний цикл програми та будує графічний інтерфейс програми використовуючи засоби бібліотеки ImGui. Для цього, викликає метод Design кожного відкритого вікна, що налаштовуються вміст цих вікон. Ініціалізує ImGui та виконує перестановку кадрів (swap buffers), як і перестановку контекстів для створення окремих вікон. Після цього виконує накопичені команди та оновлює час кадру. GUI будує випадаюче меню програми.

6.6. Вимірювання часу та логування (Time & Log)

Класс Time містить тільки статичні методи вимірювання часу. Використовується для отримування інформації про кількість кадрів, опрацьованих за секунду, або час, необхідний на опрацювання одного кадру тощо.

Клас Log є обгорткою над стандартним потоком виводу у консоль. Додає інформацію про контекст повідомлення та його важливість, тим самим уніфікуючи вивід інформації у консольне вікно. Для роботи з логером потрібно створити його екземпляр за допомогою статичного метода For, що приймає контекст як шаблонний параметр (рисунок 6.5). Після цього отриманий екземпляр можна використовувати для виведення різноманітних повідомлень. Методи Error, Warning, Information приймають довільну кількість параметрів. Усі передані параметри будуть виведені послідовно, як можна бачити на рисунку 6.6.

```
class SceneObjectInspectorWindow : Window, MoveCommand::IHolder {
    const Log log = Log::For<SceneObjectInspectorWindow>();
```

Рисунок 6.5

```
log.Error("Invalid SceneObject type passed: ", t->GetType());
//std::cout << "[" << "Error" << "]" << "(" << class SceneObjectInspectorWindow
// << ")" << "Invalid SceneObject type passed: " << t->GetType() << std::endl;
```

Рисунок 6.6

6.6. Визначення позиції (Position detection)

Визначення положення обличча та очей необхідно для вводу ракурсу спостереження у програму через камеру. Цей ракурс модифікує стартовий ракурс, який може бути введений вручну у відведеному місці у програмі (Рисунок 7.1.4 б). Цей функціонал може бути вимкнений у меню програми (Рисунок 7.1.8). Клас PositionDetector містить метод ініціалізації, що завантажує моделі для комп'ютерного бачення та перевіряє наявність підключення до камери та методи початку і кінця роботи розпізнавача. Основний цикл розпізнавача виконується у окремому потоці,

керування яким виконується через виклик методів початку та кінця роботи розпізнавача. Результатом роботи розпізнавача є набір змінних, що відповідають за позицію обличча та очей у просторі відносно камери. Взаємодія між потоками відбувається через набір атомічних полей. Детектор записує оновлені дані у поля, а функція CustomRenderFunc зчитує ці дані та модифікує властивості камери, що зображено на рисунку 6.7. Таким чином, у програмі виконується два цикли незалежно один від одного, з тим виключенням, що цикл опрацювання позиції користувача може бути зупинено чи відновлено з головного циклу через випадаюче меню програми.

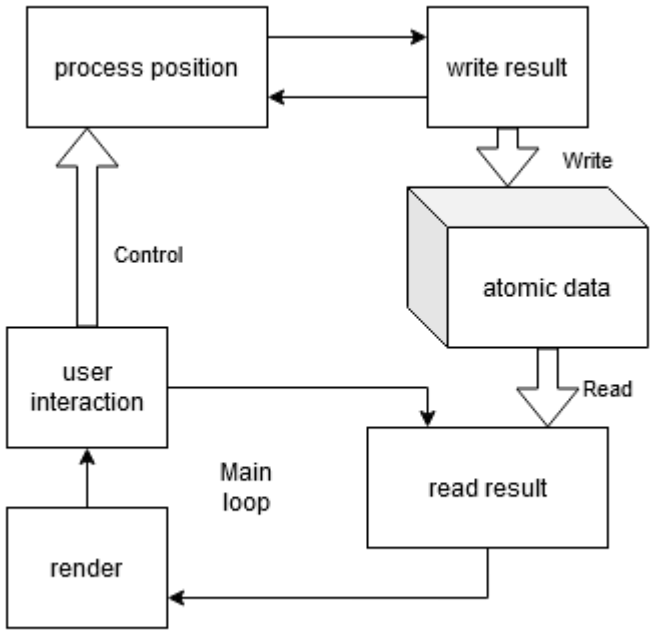


Рисунок 6.7 – Взаємодія потоків детектора позиції та головного потоку

6.6. Файлова система (FileManager)

Робота з файловою системою відбувається у файловому та текстовому режимах. Текстовий режим представляє з себе роботу з Json файлами. Те, який режим роботи буде обраний залежить від розширення файлу, що вказаний для збереження чи зберігання. Файловий менеджер працює як відносними, так і з абсолютними шляхами до файлів або папок.

Для збереження та завантаження об’єктів сцени було реалізовано набір класів, що відповідають за сериалізацію, та десериалізацію об’єктів, як видно на рисунку 6.8 де obstream, ojstream – сериалізують об’єкт у потік байтів,
ibstream, ijstream – десериалізують об’єкт з потоку байтів.

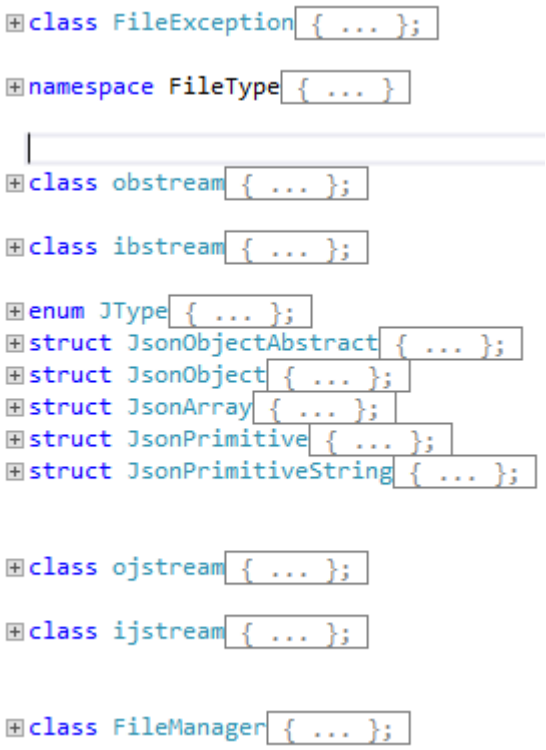


Рисунок 6.8 – вміст файлу FileManager.hpp

obstream та ibstream відповідають за байтому конвертацію. Вони є дуже простими у своїй реалізації, так як C++ дозволяє працювати з усіма об’єктами як з масивами байтів, що робить роботу з бінарними файлами дуже швидкою. Фрагмент бінарної серіалізації зображений на рисунку 6.9. Десериалізація відбувається аналогічно, з тією відмінністю, що не всі поля є публічно доступними і необхідно передавати callback функції для того, щоб їх встановлювати, що продемонстровано на рисунку 6.10.

Робота з Json форматом відбувається у два етапи:

- Файл – Json об’єкт;
- Json об’єкт – об’єкт сцени.

Зчитування об’єктів з файлу будує дерево з Json об’єктів. Ці об’єкти примітивні та мають єдину функцію – представити об’єкт через набір рядкових пар ключ –

значення, де ключем є назва поля, а значенням – інший Json об’єкт, або рядок. Як видно з рисунку 6.8, існує один базовий абстрактний Json об’єкт, та набір об’єктів, що представляють конкретні структури даних.

```
class obstream {
    std::vector<char> buffer;
public:
    template<typename T>
    void put(const T& val) {
        for (size_t i = 0; i < sizeof(T); i++)
            buffer.push_back(((char*)&val)[i]);
    }
    template<>
    void put<std::string>(const std::string& val) { ... }
    template<>
    void put<SceneObject>(const SceneObject& so) {
        put(so.GetType());
        put(so.Name);
        put(so.GetLocalPosition());
        put(so.GetLocalRotation());
```

Рисунок 6.9 – фрагмент класу obstream

Функції, що збирають об'єкти виглядають схоже до своїх бінарних родичів, але у додаток обов'яково вказують ім'я поля, яке треба зчитати, але робити це можна у довільному порядку (Рисунок 6.11).

```
case MeshT:
{
    auto o = start<Mesh>();
    read(&o->Name);
    read(std::function([&o](glm::vec3 v) { o->SetLocalPosition(v); }));
    read(std::function([&o](glm::fquat v) { o->SetLocalRotation(v); }));
    readArray(std::function([&o](glm::vec3 v) { o->AddVertice(v); }));
    readArray(std::function([&o](size_t a, size_t b) { o->Connect(a, b); }));
    readChildren(o);
    return o;
}
```

Рисунок 6.10 – фрагмент класу ibstream

```
case MeshT:
{
    auto o = start<Mesh>();
    get(j, "name", o->Name);
    get(j, "localPosition", std::function([&o](glm::vec3 v) { o->SetLocalPosition(v); }));
    get(j, "localRotation", std::function([&o](glm::fquat v) { o->SetLocalRotation(v); }));
    getChildren(j, "children", o);
    getArray(j, "vertices", std::function([&o](glm::vec3 v) { o->AddVertice(v); }));
    getArray(j, "connections", std::function([&o](size_t a, size_t b) { o->Connect(a, b); }));
    return o;
}
```

Рисунок 6.11 – фрагмент класу ifstream

7. РОБОТА КОРИСТУВАЧА З ПРОГРАМНОЮ СИСТЕМОЮ

7.1. Графічний інтерфейс програми

Користувацький інтерфейс представляє з себе головне вікно програми та багато дочірніх вікон (Рис.7.1.1). Дочірні вікна можуть бути прикріплені\відкріплені через перетягування вікна (його заголовка) за допомогою миші у бажану позицію, керуючись візуальними спрямовучами, переміщені, змінені за розміром, перетворені на вкладинки, тощо. Під час перетягування вікна з'являються візуальні підсказки, що допомагають правильно розмістити вікно (Рис.7.1.2). Налаштування дочірніх вікон зберігається при виході з програми, тож користувач може налаштувати інтерфейс до свого смаку. Також присутнє консольне вікно для виводу інформації про помилки чи попередження (Рис.7.1.3).

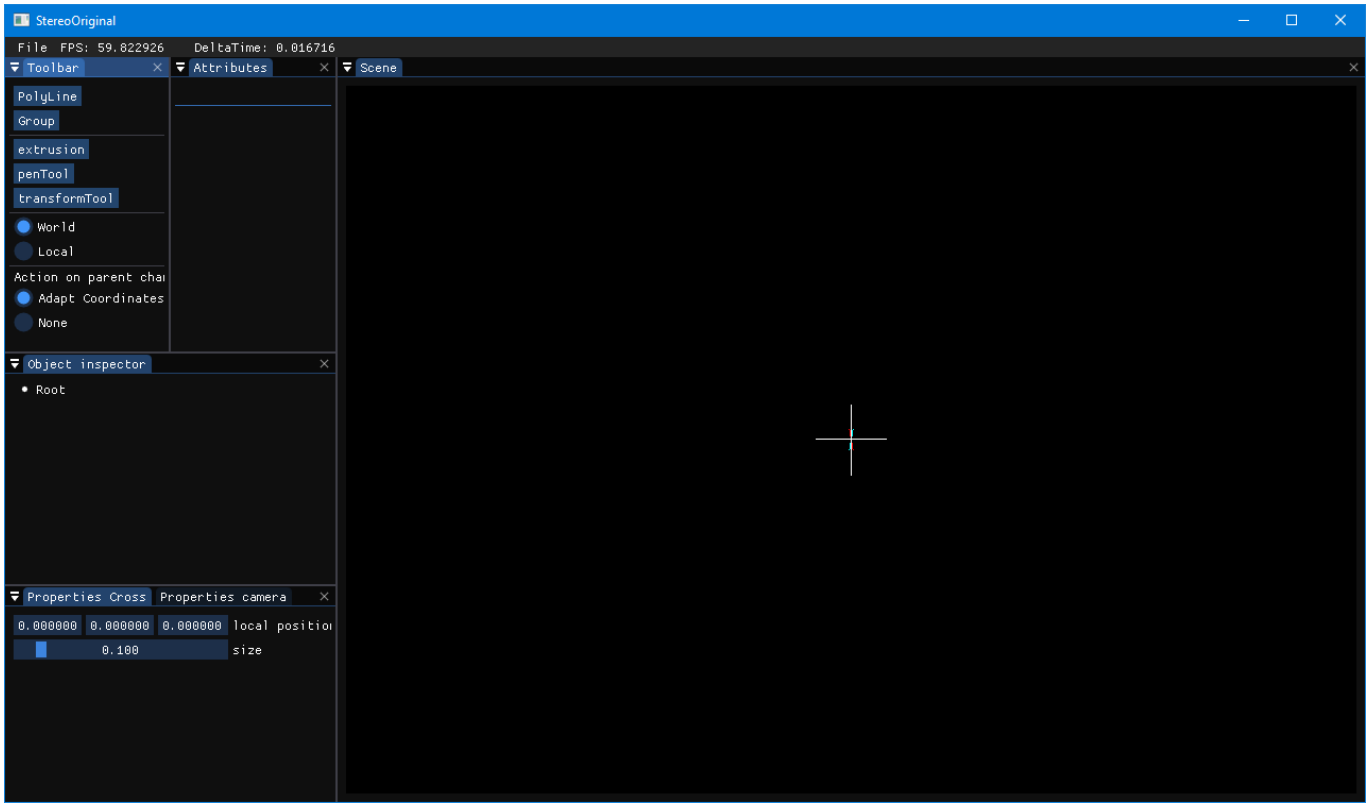


Рисунок 7.1.1 – Головне вікно програми

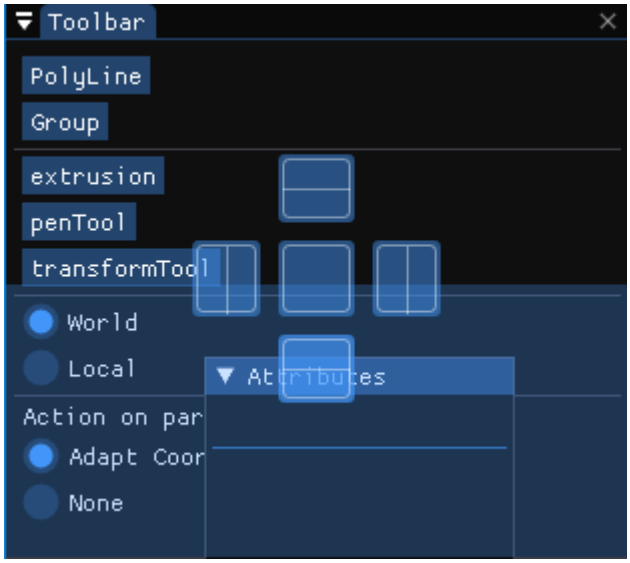


Рисунок 7.1.2 – Процес перетягування вікон

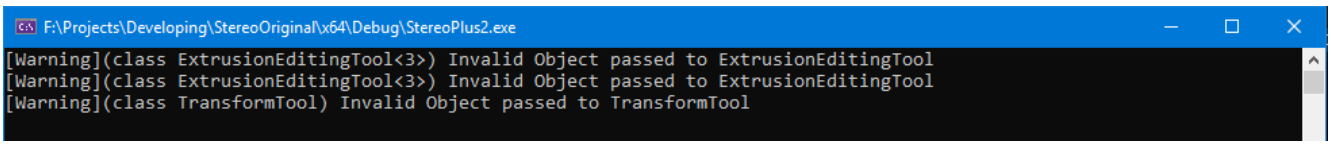
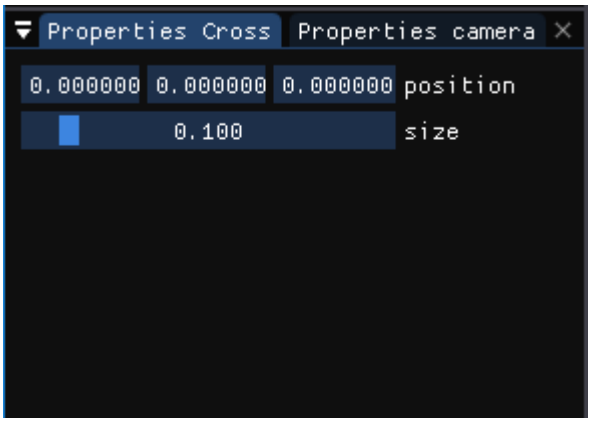
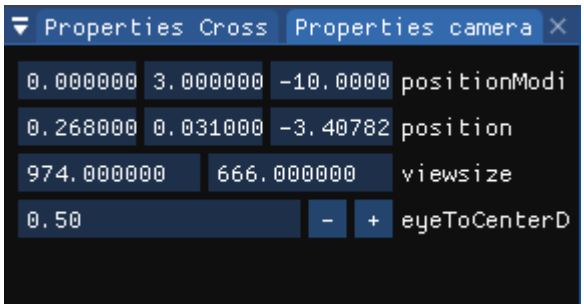


Рисунок 7.1.3 – Консольне вікно з попередженнями

Інформація про властивості об’єктів сцени доступна для читання, та редагування у вікнах властивостей (Рис.7.1.4). Властивості представляють з себе поля об’єктів, які презентують обрані вікна.



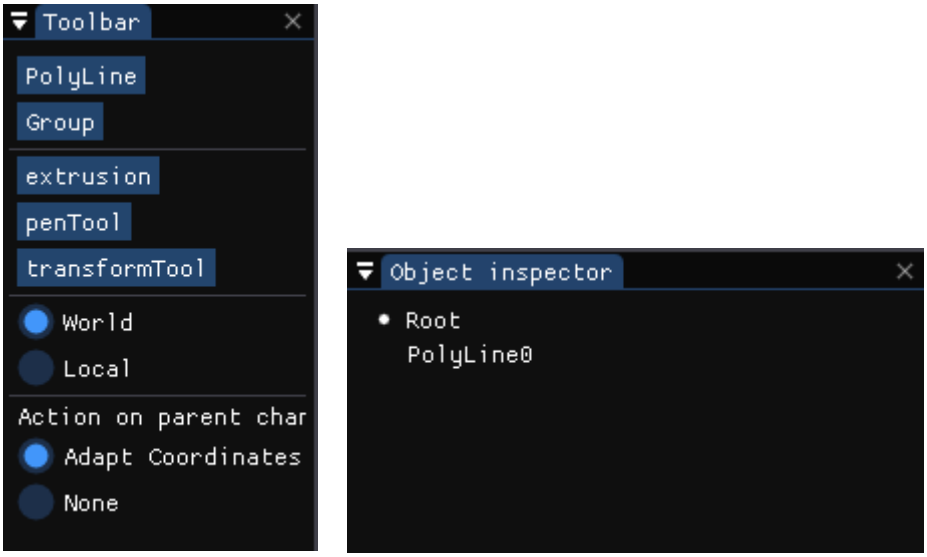


(а) (б)

Рисунок 7.1.4 – Вікна властивостей курсора (а) та камери (б)

Вікно інструментів містить інструменти для створення (до розділювача) та редагування (після розділювача) об’єктів сцени (Рис.7.1.5). Створення об'єкту відбувається натисканням на інструмент з назвою, що відповідає типу об'єкта. Створення тип об’єкту додає базовий (порожній) об’єкт до кореневої групи (Root)

сцени. Після другого розділювача зніходиться перемикач світового та локального режимів координат (Рисунок 7.1.5).



(а) (б)

Рисунок 7.1.5 – Вікно інструментів (а) та створений об'єкт Ламана (б)

Обрання інструмента редагування змінює вміст вікна атрибутів (Рис.7), відображаючи необхідні для роботи з обраним інструментом властивості.

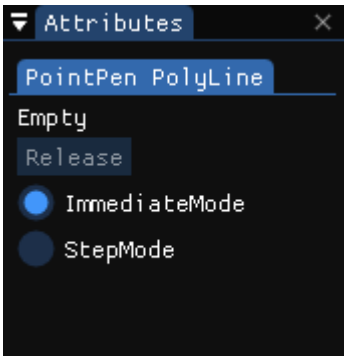


Рисунок 7.1.6 – Вікно атрибутів з обраним інструментом малювання ламаних (penTool)

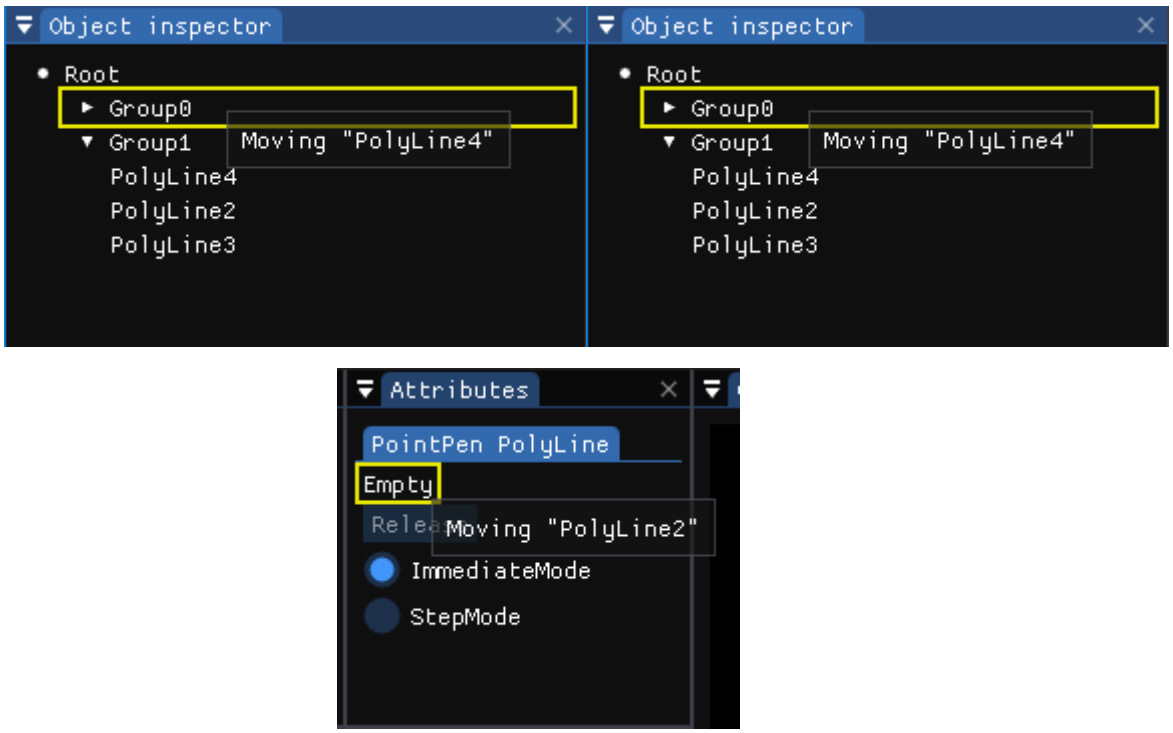
Для роботи з інструментами обраний об'єкт потрібно перетягнути з вікна інспектора об'єктів у поле об'єкта бажаного інструмента (Рис.7.1.7).

Вікно інспектора об'єктів відображає дерево об'єктів сцени з єдиним кореневим об'єктом (Root) (Рис.7.1.5, Рис.7.1.7). Об’єкти поділяються на два види: об’єкти-групи (Group, надалі група) та кінцеві об’єкти (Leaf), що представляють конкретні об’єкти як відрізок, ламана тощо. Інспектор підтримує переміщення об’єктів шляхом їхнього перетягування мишею.

Об’єкти можуть бути вставлені перед, за, та у середині об’єкту. Для цього необхідно відповідно перетягнути об’єкт на верхню частину об’єкту у бажаній позиції для вставки об’єкта перед ним, нижню – для вставки за об’єктом та у центр для переміщення у середину об’єкта. Переміщення у середину об’єкта підтримують тільки групи. За переміщення об’єкта до групи об’єкт буде розміщено на першій позиції у середині групи. Об’єкти можуть бути переміщенні у бажану позицію всередині групи за умови, що група розкрита.

Варто зауважити, що під час переміщення об’єкта між групами відбувається конвертування координат для того, щоб зберегти світові перетворення об’єкта. Дану функціональність можна вимкнути, вказавши дію під час зміну батьківського об’єкта у вікні інструментів (Рисунок 7.1.5).

Присутня можливість розкриття та згортання об’єктів-груп для спрощення організації робочої частини об’єктів сцени. Під час переміщення об’єкту біля курсору відображується назва об’єкту, що перетягується, а місця у інтерфейсі, куди може бути вставлено об’єкта обводяться жовтим контуром (Рис.7.1.7).



(а) (б)

Рисунок 7.1.7 – Перетягування об'єку сцени у згорнуту групу (а) та у інструмент малювання ламаних (б)

Меню програми досить пропонує 6 пунктів: збереження сцени, завантаження сцени, закриття сцени, перемикач багаторакурсності, перемикач FPS та вихід із програми (Рис.7.1.8).

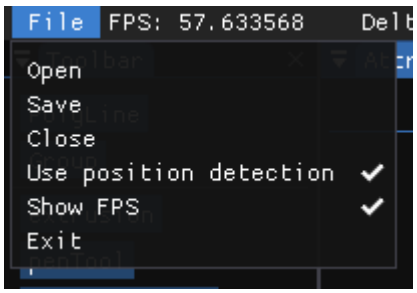


Рисунок 7.1.8 – Меню програми «Файл»

Відображення сцени відбувається у вікні відображеня сцени (Scene). Зображення у цьому вікні зберігає свої пропорції. За зразок куту зору обрано вісь Y, тож за зміни розмірів цього вікна зображення буде віддалятися чи наближуватися (кут зору по горизонталі збільшуватиметься або зменшуватиметься відповідно) за зміни висоти вікна, та залишатиметься незмінним за зміни ширини вікна.

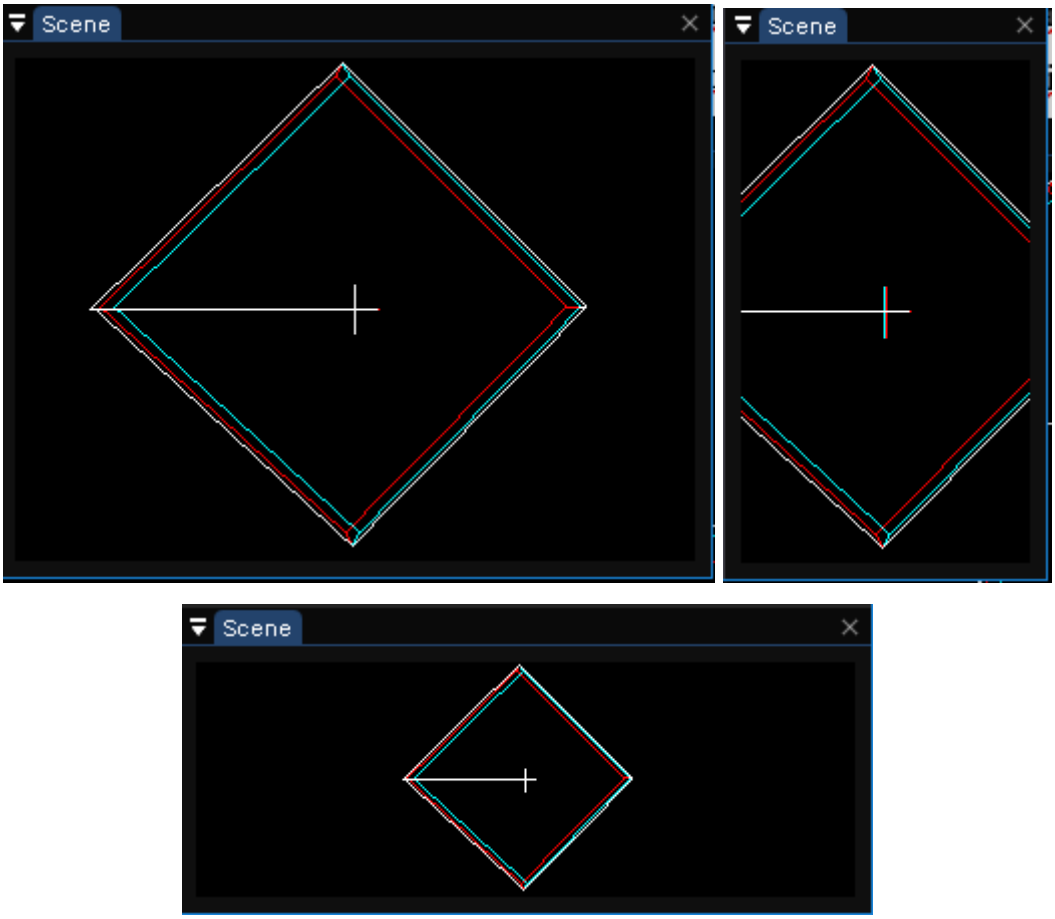


Рисунок 7.1.9 – Вікно відображення сцени зі зміненим розміром вікна у різних пропорціях

Курсор сцени представляє з себе перехрестя, що знаходиться у тому ж просторі, що і об'єкти сцени (Рис.7.1.9). Керування курсором здійснюється наступним чином:

- За допомогою NumPad клавіш: 1 – наближує, 9 – віддаляє, 4 – вліво, 6 – вправо, 8 – вгору, 2 – вниз. Також можна зменшити розмір курсору за допомогою 7 та збільшити за допомогою 3. Підтримує сповільнення курсору клавішою Ctrl;
- Клавішами стрілками. Підтримує сповільнення курсору клавішою Ctrl;
- Простий режим керування мишою, що здійснює рух курсору у площині, парілельній площині зображення камери. Підтримує сповільнення курсору клавішою Ctrl;
- Блокуючий режим керування мишою, що здійснює рух курсору у площині, або вздовж прямої, обмеженої користувачем. Обмеження руху відбувається блокуванням вісей. Так Alt блокує вісь Z, Ctrl – Y, Shift – X. Рух здійснюється тільки якщо хоча б ода з вісей заблокована. Якщо заблокувати усі три вісі, то курсор рухатися не буде.

Положення, та розмір курсору можна побачити та відредагувати у вікні влістивостей курсора (Рис.7.1.4).

Варто зауважити, що курсор не завжди реагує рухом на згадані вище дії. Інструменти перегружають підписання курсору на ввід користувача для того, щоб над його перетвореннями можна було здійснювати різні операції.

7.2. Інструменти редагування

Після вибору інструмента, його властивості з’являються у вікні атрибутів, де відбувається подальша взаємодія з інструментом. Для того, щоб почати роботу з інструментом необхідно прив'язати об'єкт до інструмента. Це здійснюється шляхом перетягування об'єкта з вікна інспектора об'єктів до відповідного поля інструмента (Рис.7.1.7). Після присвоювання об’єкта інструменту рух курсора починає впливати

на цей об'єкт. Натискання клавіші Escape зупиняє роботу з інструментом ті відв’язує об’єкт від інструмента.

Реалізована робота у світових та локальних координатах, при цьому курсор займає відповідні поворот та положення в залежності від батьківського об’єкта.

7.2.1. Малювання ламаних

У покроковому режимі (Рис.7.2.1) рух курсора переносить майбутню точку ламаної у позицію курсора. Натискання Enter закріплює майбутню точку у поточній позиції та створює нову майбутню точку. У моментальному режимі точки закріплюються по мірі руху курсора автоматично. Таким чином можна малювати ламані, подібні до кривих (Рис.7.2.2). Натискання Escape видаляє майбутню точку та відв’язує об’єкт у його поточному вигляді у обох режимах.

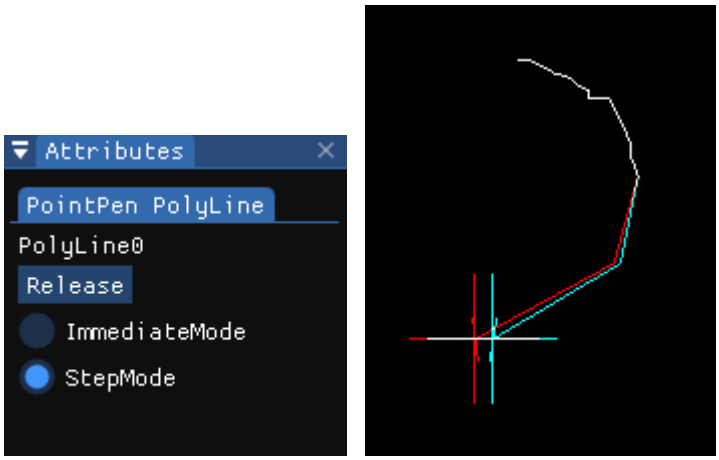


Рисунок 7.2.1 – Покроковий режим малювання ламаних

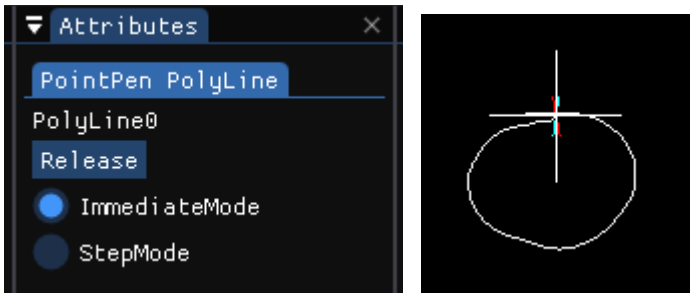


Рисунок 7.2.2 – Моментальний режим малювання ламаних

Режими можна чергувати. Таким чином можна намалювати частину ламаної у моментальному режимі, далі частину у покроковому і так далі.

7.2.2. Вичавлювання ламаних

Вичавлювання відбувається так само як і малювання ламаних. Обирається бажана ламана, режим, створюється об’єкт куди буде збережено вичавлений об’єкт, та починається очікування вводу користувача. Усе керування здійснюється аналогічно до малювання ламаних. На рисунках 7.2.3 та 7.2.4 зображено вичавленні ламані зображені на рисунках 7.2.1 та 7.2.2 відповідно.

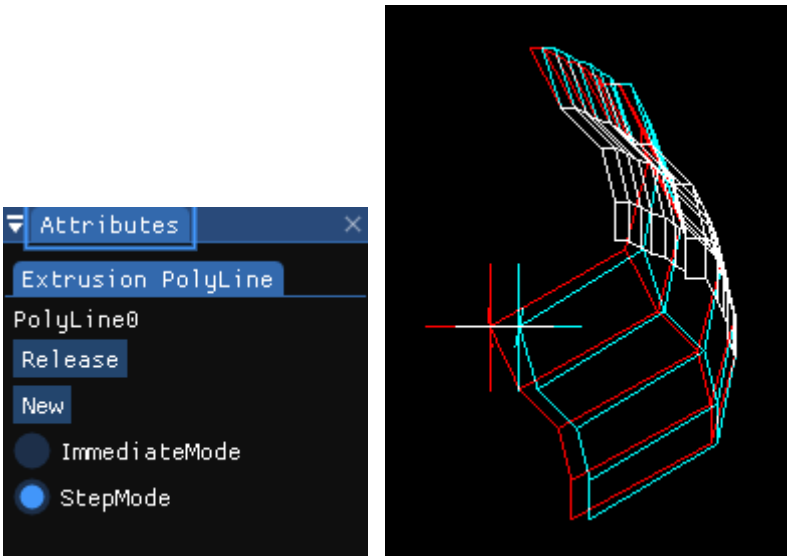


Рисунок 7.2.3 – Покроковий режим вичавлювання ламаної

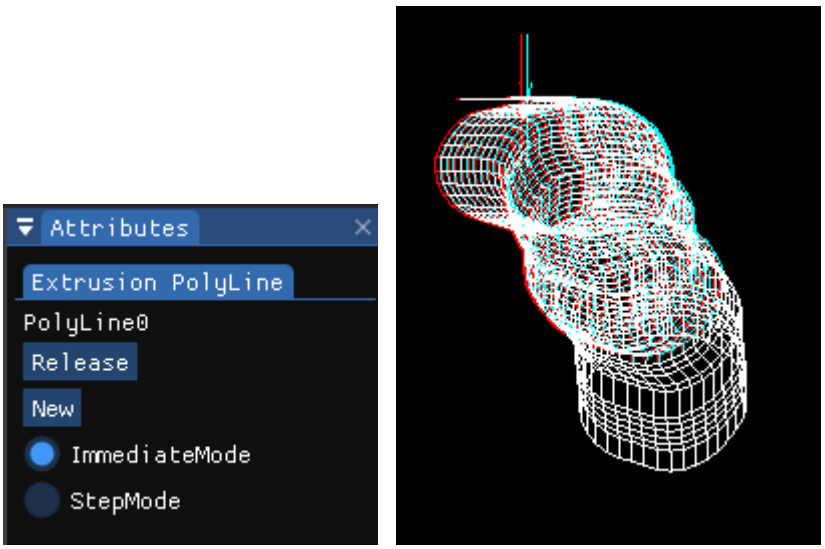


Рисунок 7.2.4 – Моментальний режим вичавлювання ламаної

Реалізовано оптимізацію малювання у моментальному режимі (Immediate mode), що мінімізує кількість створених відрізків через відстеження напрямку руху курсору. Таким чином, якщо рухати курсор по прямій, то на цьому проміжку буде створено тільки один відрізок. Також відбувається фільтрація на мінімальну довжину відрізка, що запобігає створенню великої кількості відрізків дуже малої довжини, що утворюються через похибку точності типу float, та інших факторів.

7.2.3. Перетворення об'єктів

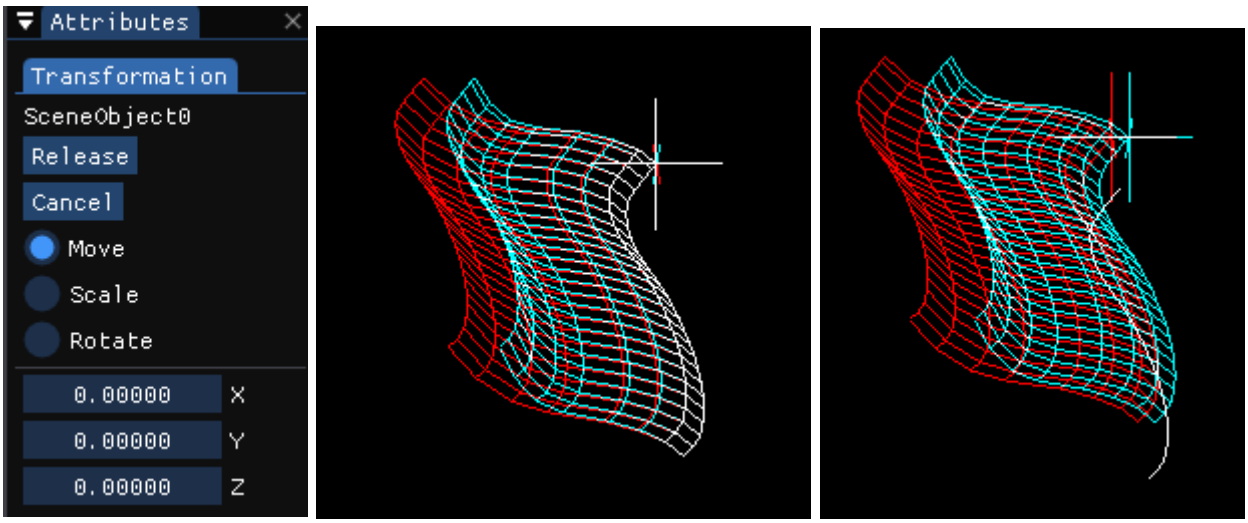
Геометричні перетворення об’єктів відбуваються за допомогою інструмента трансформування (Transform). Реалізовано три режими роботи: Перенесення, масштабування та поворот. Натиснення клавіші Escape або кнопки Cancel відміняє перетворення, та повертає об’єкт у початкове положення. Натиснення клавіші Enter або кнопки Release застосовує зміни до об’єкта та відв’язує його від інструмента.

Для переміщення необхідно обрати об’єкт та перемістити курсор або встановити позицію курсора у вікні властивостей курсора (Рис.7.1.4). Переміщення об’єкта відображається у вікні інструмента у вигляді перетягувачів. «Перетягувач» може бути переведений у режим текстового поля за допомогою кліка на ньому з натиснутою клавішою Ctrl або подвійного натиску лівої кнопки миші. У такому випадку можна вказати бажане число з клавіатури.

Об’єкт буде переміщуватися разом з курсором. На рисунку 7.2.5 зображено об’єкт перед переміщенням (а), та після переміщення уперед (в) (вісь Z).

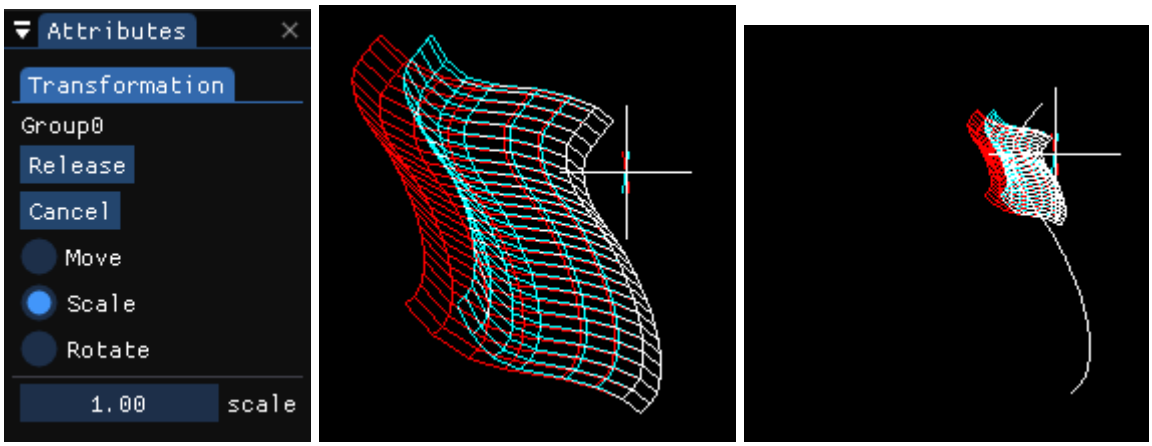
Для масштабування об’єкта необхідно обрати об’єкт, за допомогою курсора вказати центр масштабування та, користуючись перетягувачем масштабу встановити бажаний масштаб об’єкту.

Для масштабування навколо довільної точки необхідно створити груповий об’єкт, перемістити його у точку майбутнього центра перетворення та помістити у нього бажаний об’єкт зі збереженням його світових координат. Далі необхідно масштабувати створений груповий об’єкт. На рисунку 7.2.6 зображено зменшення масштабу об’єкта до значення 0.3 відносно точки, заданої груповим об’єктом.



(а) (б) (в)

Рисунок 7.2.5 – Атрибути інструмента переміщення (а), об’єкт перед переміщенням (б) та об’єкт після переміщення (в)

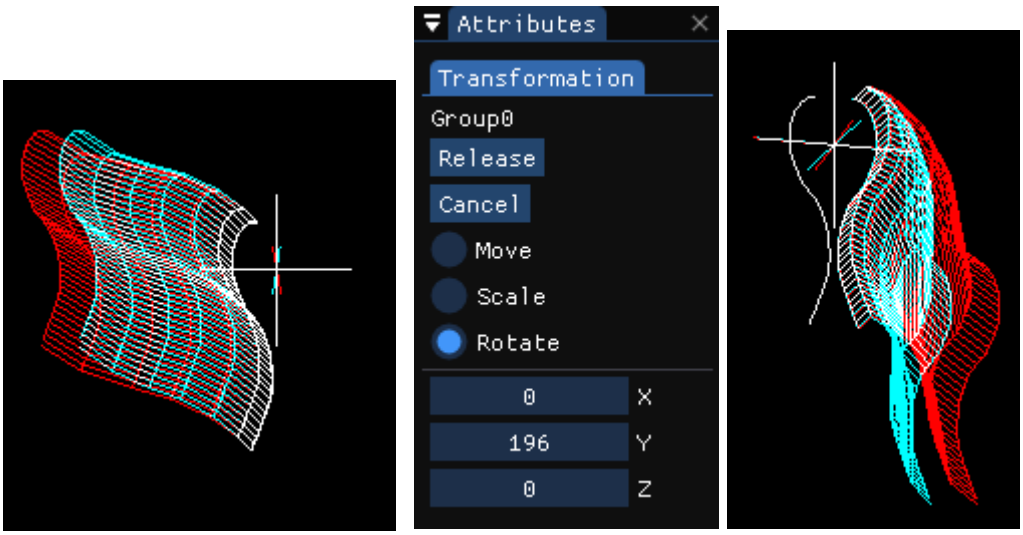


(а) (б) (в)

Рисунок 7.2.6 – Атрибути інструмента масштабування (а), об'єкт перед масштабуванням (б) та після (в)

Поворот об'єкта відбувається схожим чином. Обирається об'єкт, обирається центр вісі обертання за допомогою групового об'єкта якщо потрібно та за допомогою вже знайомого «перетягувача» обирається кут повороту об’єкта навколо обраної осі. Кут повороту вказується у градусах. На рисунку 7.2.7 зображено поворот об’єкта навколо вісі Y на 196 градусів.

Варто зазначити, що курсор повертається разом з об’єктом, тож курсор також вказує на вісі повороту.



(а) (б) (в)

Рисунок 7.2.7 – Об'єкт перед поворотом (а), атрибути інструмента поворота (б) та повернутий об'єкт (в)

7.3. Робота з файловою системою

Робота з файловою системою представляє з себе вікна завантаження та збереження сцени (Рис.7.3.1). Зовнішньо вони мало відрізняються. Поле Шлях (Path) відображає поточний шлях. Шлях можна відредагувати і перейти до нього натиснувши на кнопку Submit. На панелі файлів можна бачити посилання на батьківську директорію (..), папки у квадратних дужках та файли, назви яких закінчуються назвою їх типу. У полі File виводиться шлях до обраного файлу. Це поле також можна редагувати і вказати шлях до файлу наприклад. Програма розуміє абсолютний та відносний шляхи. Для відкриття файлу потрібно вказати файл на натиснути кнопку Open. Для збереження потрібно відкрити вікно збереження, обрати шлях до файлу та натиснути Save.

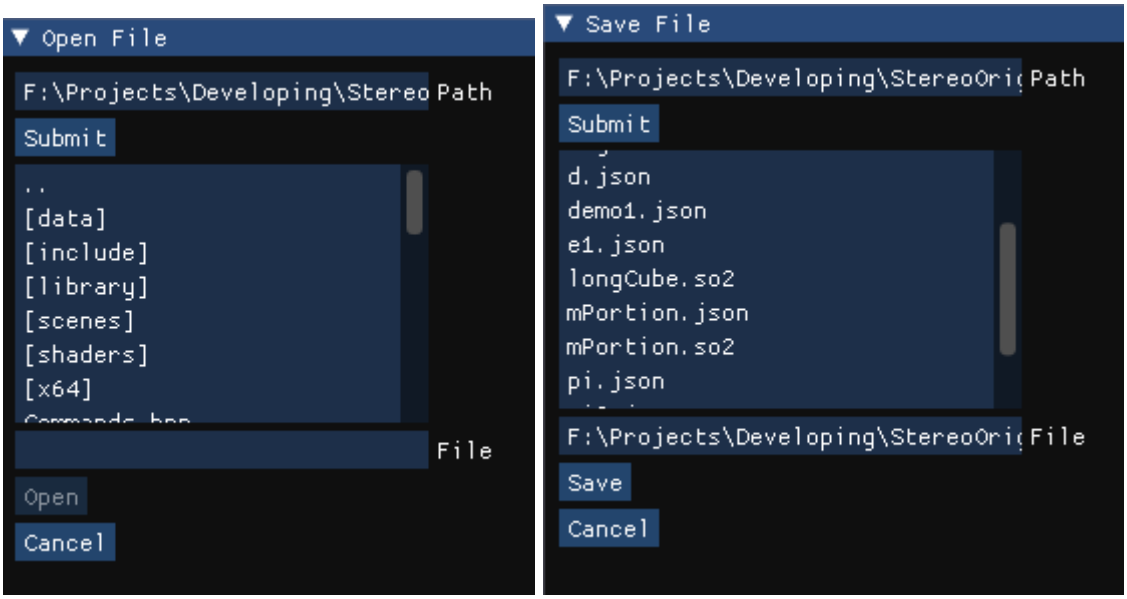


Рисунок 7.3.1 – Вікно завантаження (зліва) та збереження (справа) сцени

Підтримується два формати: so2, що є бінарним; та json, що є розповсюдженим форматом зберігання даних за моделлю ключ-значення. Тип файлу визначається за розширенням файлу.

Бінарний ввід\вивід

Реалізовано клас для сериалізації об’єктів сцени у бінарний вигляд (Рис.7.3.2). Даний метод є швидшим за Json сериалізацію, та утворені файли займають менше місця. На рисунках 7.3.2 та 7.3.3 зображені файли створені цією програмою з однієї сцени у форматах so2 та json відповідно.

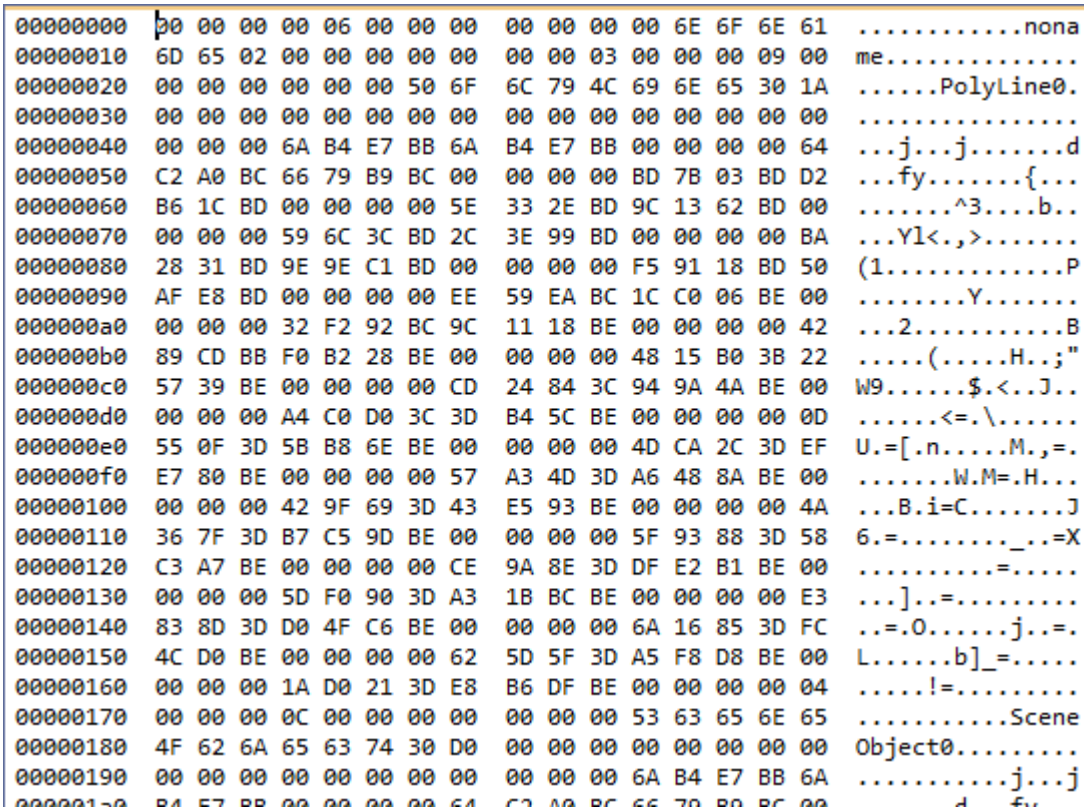


Рисунок 7.3.2 – so2 файл, відкритий у бінарному редакторі Visual Studio

Json ввід\вивід

Реалізовано клас для сериалізації об’єктів сцени у json вигляд (Рис.7.3.3). Даний метод є повільнішим за бінарну сериалізацію, та утворені файли займають більше місця. Проте утворені файли є читабельні для людини, вони можуть бути легко модифіковані чи навіть створені у будь якому текстовому редакторі і є зручними у візуалізації структури сцени, так як є легко форматованими у багатьох текстових редакторах чи IDE (Рис.7.3.4).

{"type":0,"name":"noname","children":[{"type":3,"name":"PolyLine0","points"
:[[0,0,0],[-0.00707107,-0.00707107,0],[-0.0196239,-0.0226409,0],[-0.0321004
,-0.0382603,0],[-0.0425295,-0.0551945,0],[-0.0460018,-0.0748256,0],[-0.0432
517,-0.0945408,0],[-0.0372486,-0.113616,0],[-0.0286073,-0.131592,0],[-0.017
9378,-0.148505,0],[-0.00627247,-0.164745,0],[0.00537363,-0.180996,0],[0.016
1308,-0.197855,0],[0.0254825,-0.215531,0],[0.0349932,-0.233125,0],[0.042185
1,-0.25177,0],[0.0502046,-0.270086,0],[0.0570366,-0.288859,0],[0.0623076,-0
.308149,0],[0.0666873,-0.327662,0],[0.0696312,-0.347434,0],[0.070771,-0.367
398,0],[0.0690992,-0.387328,0],[0.0649842,-0.406837,0],[0.0545324,-0.423772
,0],[0.0395051,-0.436942,0]]},{"type":4,"name":"SceneObject0","vertices":[[
0,0,0],[-0.00707107,-0.00707107,0],[-0.0196239,-0.0226409,0],[-0.0321004,-0
.0382603,0],[-0.0425295,-0.0551945,0],[-0.0460018,-0.0748256,0],[-0.0432517

Рисунок 7.3.3 – json файл створений програмою

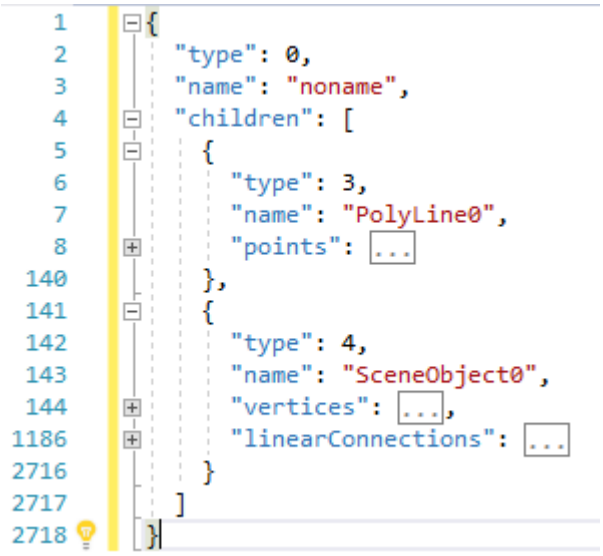


Рисунок 7.3.4 – Відформатований json файл, відкритий у Visual Studio

7.4. Багаторакурсне спостереження

Багаторакурсне спостереження дозволяє користувачу змінювати ракурс зображення через зміну положення обличча у просторі перед камерою. На рисунку 7.4.1 зображено кілька сикриншотів сцени з різним положенням обличча відносно камери:

- (г) – по центру;
- (а) – вище центру;
- (е) – нище центру;
- (в) – зліва від центру;
- (ґ) – справа від центру;
- (б) – блище до камери;
- (д) – далі від камери.

Через малий розмір рисунку скриншоти (б) та (д) майже нічим не відрізняються. Вимкнути, або увімкнути відстеження можна через меню програми File -> Use position detection (Рис.7.1.8).

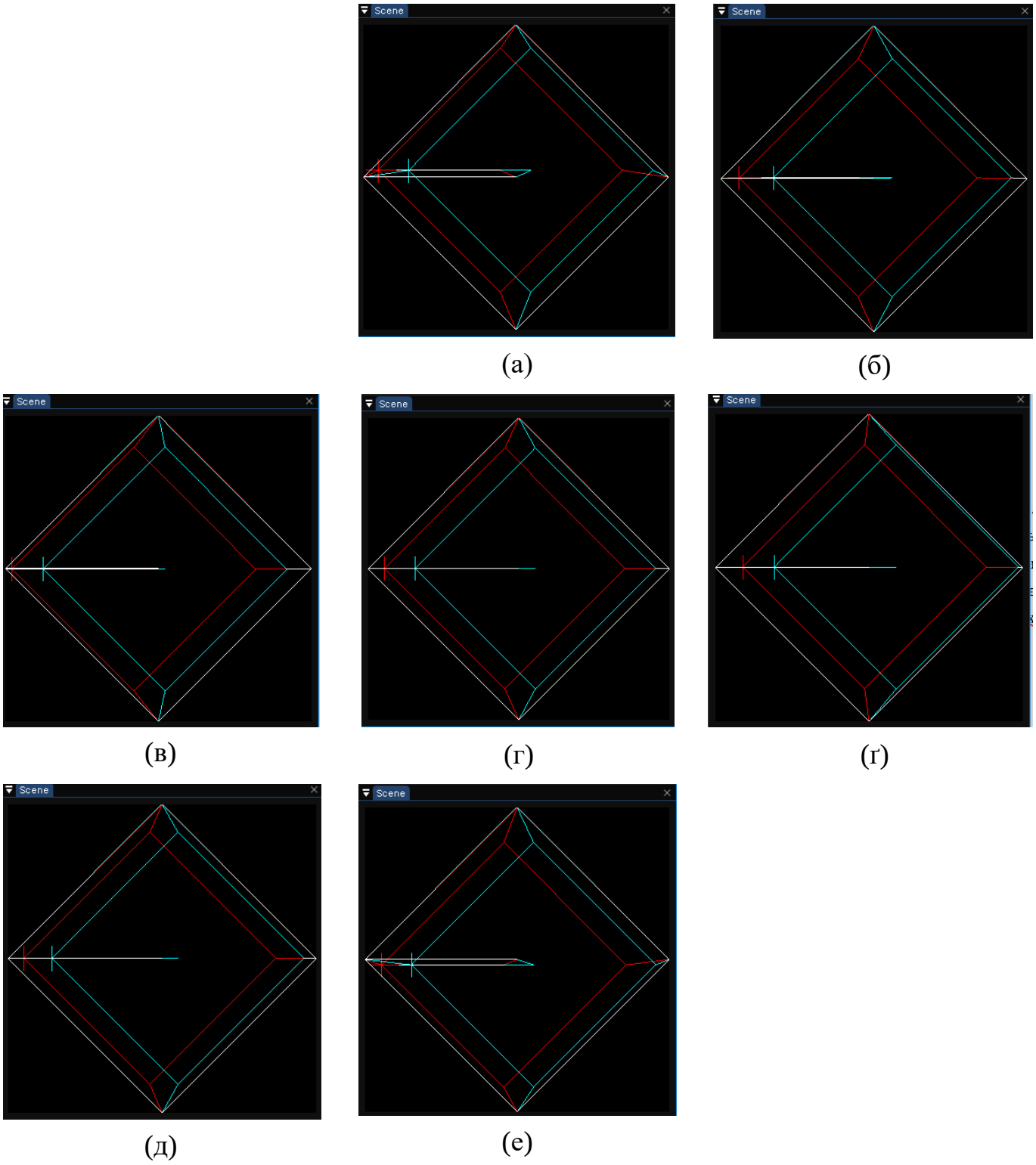


Рисунок 7.4.1 – Перетворення зображення залежно від положення користувача перед камерою

ВИСНОВКИ

В ході роботи було розроблено програмний продукт для реалізації основних тривимірних процедур взаємодії між реальним і віртуальним світом на основі теорії стереооператорів для точково-скелетних 3D-зображень, з можливістю введення зворотного зв'язку по стереоракурсу.

Реалізовані функції:

- Конвертація між тривимірними координатами об’єкта, та стереокоординатами монітора;
- Щуп (тривимірний курсор);
- Малювання кривих та ламаних;
- Малювання кривими та ламаними;
- Переміщення, поворот та масштабування об’єктів;
- Локальна та глобальна системи координат;
- Групування об’єктів та перетворень;
- Адаптація перетворень за переміщення об’єкта між груповими об’єктами;
- Модифікація стереокоординат за допомогою відстеження позиції користувача;
- Завантаження\збереження об’єктів у файл.

Реалізована функціональність дозволяює більш адекватно та реалістично відображати точково-скелетні зображення та працювати з ними.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Daneshmand M. 3D Scanning: A Comprehensive Survey / M. Daneshmand, A. Helmi, E. Avots, F. Noroozi, F. Alisinanoglu, H. S. Arslan, J. Gobrova, R. E. Haamer, C. Ozcinar, G. Anbarjafari // arXiv — 1801.08863v1, 24 Jan 2018, 18 p.
2. Stankovic Z. 4D flow imaging with MRI / Z. Stankovic, B. D. Allen, J. Garchia, K. B. Jarvis, M. Markl // Cardiovasc Diagn Ther. — 2014 Apr; 4(2): 173–192 pp:
3. Kwan S. A Fast Method for 3D CFD Modeling of a Long River Reach / S. Kwan, J.A. Vasquez, R.G. Millar, P.M. Steffler // World Environmental and Water Resources Congress — 2011, 8 p.
4. Pontin D.I. Observable Signatures of Energy Release in Braided Coronal Loops / D. I. Pontin, M. Janvier, S. K. Tiwari, K. Galsgaard, A. R. Winebarger, and J. W. Cirtain // University of Dundee, Nethergate, Dundee — DD1 4HN, UK, March 8 2017 — 10 p.
5. Валюс Н. А. Стереоскопия. — М. : Наука, 1962. – 378 с.
6. Груц Ю.Н. Стереоскопическая машинная графика. — Киев: Наукова думка, 1989. — 160 с.
7. Gruts Yu.N. Stereoscopic operators and its application / Gruts Yu.N., Jung-Young Son // Proccedings of The 6th International Workshop on 3-D Imaging Media Technology and The 5th Photonic Information Processing Conference. — 2000. — Vol.6, N1, pp. 34-38.
8. Груц Ю.Н. Операторний метод стереоперетворень в комп’ютерній графіці / Материалы 6-й Международной научно-технической конференциии: Информационные системы и технологии ИСТ-2017, посвященной 80-летию В.В. Свиридова 11–16 сентября 2017. — Коблево, Украина. — Харьков. — С. 37-38.

9. Viola P. Rapid Object Detection using a Boosted Cascade of Simple Features / P. Viola, M. Jones // Accepted conference on computer vision and pattern recognition, 2001 – 9 p.
10. Mairal J. Convolutional Kernel Networks / J. Mairal, Koniusz, Z. Harchaoui, C. Schmid // Neural Information Processing Systems Conference. — 2014, 9 p.
11. Dantam N. 1 Quaternion Computation / N. Dantam // Institute for Robotics and Intelligent Machines, Georgia Institute of Technology, Atlanta, GA, USA — 2014, 9 p.
12. Troelsen A. COM and .NET Interoperability / A. Troelsen // Apress; Softcover reprint of the original 1st ed. Edition — April 20, 2002, 769 p.
13. Nesteruk D. Design Patterns in Modern C++: Reusable Approaches for Object-Oriented Software Design / D. Nesteruk // Apress — 1st ed. Edition, April 19, 2018, 327 p.
14. Pozrikidis C. Introduction to C++ Programming and Graphics / C. Pozrikidis // Springer — ISBN-13: 978-0387689920, 2007, 384 p.
15. Kessenich J. OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.5 with SPIR-V / J. Kessenich, G. Sellers, D. Shreiner // Addison-Wesley Professional — 9th edition, July 18, 2016, 976 p.
16. Bradski G. Learning OpenCv: Computer Vision With The Opencv Library / G. Bradski, A. Kaehler // O'Reilly Media — 1st edition, October 4, 2008, 555p.

Додаток А

Реалізація процедур для побудови каркасних стерео зображень з урахуванням багаторакурсного спостереження

Специфікація

УКР.НТУУ”КПІ”ІМ.ІГОРЯ_СІКОРСЬКОГО_ТЕФ_АПЕПС_ТВ6128_20Б

Аркушів 1

Київ — 2020

Позначення	Найменування	Примітки
Документація		
УКР.НТУУ”КПІ”ІМ.ІГОРЯ_ СІКОРСЬКОГО_ТЕФ_АПЕПС_ ТВ6128_20Б 81-1	Пояснювальна записка.docx	Пояснювальна записка
Компоненти		
УКР.НТУУ”КПІ”ІМ.ІГОРЯ_ СІКОРСЬКОГО_ТЕФ_АПЕПС_ ТВ6128_20Б 12-1	main.cpp, GUI.hpp, Renderer.hpp	Модуль графічного інтерфейсу
УКР.НТУУ”КПІ”ІМ.ІГОРЯ_ СІКОРСЬКОГО_ТЕФ_АПЕПС_ ТВ6128_20Б 13-1	Додаток Б	Опис програмного модуля

Додаток Б

Реалізація процедур для побудови каркасних стерео зображень з урахуванням багаторакурсного спостереження

Текст програмного модулю

УКР.НТУУ”КПІ”ІМ.ІГОРЯ_СІКОРСЬКОГО_ТЕФ_АПЕПС_ТВ6128_20Б

Аркушів 9

Київ — 2020

```
bool CustomRenderFunc(Scene& scene, Renderer& renderPipeline, PositionDetector& positionDetector)
{
	// Modify camera posiiton when Posiiton detection is enabled.
	if (positionDetector.isPositionProcessingWorking)
		scene.camera->SetLocalPosition(
			glm::vec3(
				positionDetector.positionHorizontal / 500.0,
				positionDetector.positionVertical / 500.0,
				-positionDetector.distance / 10.0));

	// Run scene drawing.
	renderPipeline.Pipeline(scene);

	return true;
}

int main(int, char**) {
	// Declare main components.
	PositionDetector positionDetector;

	CustomRenderWindow customRenderWindow;
	SceneObjectPropertiesWindow<StereoCamera> cameraPropertiesWindow;
	SceneObjectPropertiesWindow<Cross> crossPropertiesWindow;
	SceneObjectInspectorWindow inspectorWindow;

	AttributesWindow attributesWindow;
	ToolWindow toolWindow;

	Scene scene;
	StereoCamera camera;
	Renderer renderPipeline;
	GUI gui;
	Cross cross;

	// Initialize main components.
	toolWindow.attributesWindow = &attributesWindow;
	toolWindow.scene = &scene;

	inspectorWindow.rootObject = (GroupObject**)&scene.root;
	inspectorWindow.selectedObjectsBuffer = &scene.selectedObjects;

	cameraPropertiesWindow.Object = &camera;
	scene.camera = &camera;

	if (!renderPipeline.Init())
		return false;

	gui.windows = {
		(Window*)&customRenderWindow,
		(Window*)&cameraPropertiesWindow,
		(Window*)&crossPropertiesWindow,
		(Window*)&inspectorWindow,
		(Window*)&attributesWindow,
		(Window*)&toolWindow,
	};

	gui.glWindow = renderPipeline.glWindow;
	gui.glsl_version = renderPipeline.glsl_version;

	scene.glWindow = gui.glWindow;
	scene.camera->viewSize = &customRenderWindow.renderSize;
	scene.cross = &cross;
```

```
	gui.scene = &scene;

	cross.Name = "Cross";
	if (!cross.Init())
		return false;

	crossPropertiesWindow.Object = &cross;
	gui.keyBinding.cross = &cross;
	if (!gui.Init())
		return false;

	cross.keyboardBindingHandler = [&cross, i = &gui.input]()
{ cross.SetLocalPosition(cross.GetLocalPosition() + i->movement); };
	cross.keyboardBindingHandlerId = gui.keyBinding.AddHandler(cross.keyboardBindingHandler);

	* ToolPool::GetCross() = &cross;
	* ToolPool::GetScene() = &scene;
	* ToolPool::GetKeyBinding() = &gui.keyBinding;

	if (!ToolPool::Init())
		return false;

	// Position detector doesn't initialize itself
	// so we need to help it.
	positionDetector.onStartProcess = [&positionDetector] {
		positionDetector.Init();
	};

	// Track the state of Position detector to switch it
	// when necessary.
	// Reads user position and modifies camera position when enabled.
	// Calls drawing methods of Renderer.
	customRenderWindow.customRenderFunc = [&scene, &renderPipeline, shouldUsePositionDetection
= &gui.shouldUsePositionDetection, &positionDetector]{
		if (*shouldUsePositionDetection && !positionDetector.isPositionProcessingWorking)
			positionDetector.StartPositionDetection();
		else if (!*shouldUsePositionDetection &&
positionDetector.isPositionProcessingWorking)
			positionDetector.StopPositionDetection();

		return CustomRenderFunc(scene, renderPipeline, positionDetector);
	};

	// Start the main loop and clean the memory when closed.
	if (!gui.MainLoop() |
		!gui.OnExit()) {
		positionDetector.StopPositionDetection();
		return false;
	}

	// Stop Position detection thread.
	positionDetector.StopPositionDetection();
    return true;
}



class GUI {
#pragma region Private

	const Log log = Log::For<GUI>();
	bool shouldClose = false;
```

```
	FileWindow* fileWindow = nullptr;

	//process all input: query GLFW whether relevant keys are pressed/released this frame and
react accordingly
	//-----------------------------------------------------------------------------------------
----------------
	void ProcessInput(GLFWwindow* glWindow)
	{
		input.ProcessInput();
	}

	bool CreateFileWindow(FileWindow::Mode mode) {
		auto fileWindow = new FileWindow();

		fileWindow->mode = mode;
		fileWindow->BindScene(scene);

		if (!fileWindow->Init())
			return false;

		windows.push_back((Window*)fileWindow);

		this->fileWindow = fileWindow;
		fileWindow->OnExit().AddHandler([f = &this->fileWindow]{
			(new FuncCommand())->func = [f = f] {
				delete* f;
				*f = nullptr;
				};
			});

		return true;
	}

	bool OpenFileWindow(FileWindow::Mode mode) {
		if (fileWindow != nullptr)
			fileWindow->mode = mode;
		else if (!CreateFileWindow(mode))
			return false;

		return true;
	}

	bool DesignMenuBar() {
		if (ImGui::BeginMenu("File")) {
			if (ImGui::MenuItem("Open", nullptr, false))
				if (!OpenFileWindow(FileWindow::Load))
					return false;
			if (ImGui::MenuItem("Save", nullptr, false))
				if (!OpenFileWindow(FileWindow::Save))
					return false;
			if (ImGui::MenuItem("Close", nullptr, false))
				scene->DeleteAll();

			ImGui::MenuItem("Use position detection", nullptr,
&shouldUsePositionDetection);
			ImGui::MenuItem("Show FPS", nullptr, &shouldShowFPS);

			if (ImGui::MenuItem("Exit", nullptr, false))
				shouldClose = true;

			ImGui::EndMenu();
		}

		if (shouldShowFPS) {
```

```
                    ImGui::LabelText("", "FPS: %-12f DeltaTime: %-12f", Time::GetFrameRate(),
Time::GetDeltaTime());
             }

             return true;
      }

      bool DesignMainWindowDockingSpace()
      {
             // We are using the ImGuiWindowFlags_NoDocking flag to make the parent window not
dockable into,
             // because it would be confusing to have two docking targets within each others.
             ImGuiWindowFlags window_flags = ImGuiWindowFlags_MenuBar |
ImGuiWindowFlags_NoDocking;
             ImGuiViewport* viewport = ImGui::GetMainViewport();
             ImGui::SetNextWindowPos(viewport->Pos);
             ImGui::SetNextWindowSize(viewport->Size);
             ImGui::SetNextWindowViewport(viewport->ID);
             ImGui::PushStyleVar(ImGuiStyleVar_WindowRounding, 0.0f);
             ImGui::PushStyleVar(ImGuiStyleVar_WindowBorderSize, 0.0f);
             window_flags |= ImGuiWindowFlags_NoTitleBar | ImGuiWindowFlags_NoCollapse |
ImGuiWindowFlags_NoResize | ImGuiWindowFlags_NoMove;
             window_flags |= ImGuiWindowFlags_NoBringToFrontOnFocus |
ImGuiWindowFlags_NoNavFocus;

             // Important: note that we proceed even if Begin() returns false (aka window is
collapsed).
             // This is because we want to keep our DockSpace() active. If a DockSpace() is
inactive,
             // all active windows docked into it will lose their parent and become undocked.
             // We cannot preserve the docking relationship between an active window and an
inactive docking, otherwise
             // any change of dockspace/settings would lead to windows being stuck in limbo and
never being visible.
             ImGui::PushStyleVar(ImGuiStyleVar_WindowPadding, ImVec2(0.0f, 0.0f));

             // Main window docking space cannot be closed.
             bool open = true;
             ImGui::Begin("MainWindowDockspace", &open, window_flags);

             // This place is a mystery for me.
             // Need to investigate it.
             // 2 is a magic number for now.
             {
                    ImGui::PopStyleVar();
                    ImGui::PopStyleVar(2);
             }

             // DockSpace
             ImGuiIO& io = ImGui::GetIO();
             ImGuiID dockspace_id = ImGui::GetID("MyDockSpace");
             ImGui::DockSpace(dockspace_id, ImVec2(0.0f, 0.0f), ImGuiDockNodeFlags_None);

             if (ImGui::BeginMenuBar()) {
                    if (!DesignMenuBar())
                           return false;

                    ImGui::EndMenuBar();
             }

             ImGui::End();

             return true;
      }
```

```
#pragma endregion
public:
	// It seems to be garbage collected or something.
	// When trying to free it it fails some imgui internal free function.
	ImGuiIO* io;
	const char* glsl_version;
	GLFWwindow* glWindow;
	Input input;
	KeyBinding keyBinding;
	Scene* scene;

	bool shouldUsePositionDetection = false;
	bool shouldShowFPS = true;

	std::vector<Window*> windows;
	std::function<bool()> customRenderFunc;

	bool Init()
	{
		keyBinding.input = &input;
		input.glWindow = glWindow;

		if (!input.Init() ||
			!keyBinding.Init())
			return false;

		// Setup Dear ImGui context
		IMGUI_CHECKVERSION();
		ImGui::CreateContext();
		io = &ImGui::GetIO(); (void)io;
		io->ConfigFlags |= ImGuiConfigFlags_NavEnableKeyboard;       // Enable Keyboard
Controls
		io->ConfigFlags |= ImGuiConfigFlags_DockingEnable;           // Enable Docking
		io->ConfigFlags |= ImGuiConfigFlags_ViewportsEnable;         // Enable Multi-
Viewport / Platform Windows
		//io.ConfigViewportsNoAutoMerge = true;
		//io.ConfigViewportsNoTaskBarIcon = true;

		// Setup Dear ImGui style
		ImGui::StyleColorsDark();
		//ImGui::StyleColorsClassic();

		// When viewports are enabled we tweak WindowRounding/WindowBg so platform windows
can look identical to regular ones.
		ImGuiStyle& style = ImGui::GetStyle();
		if (io->ConfigFlags & ImGuiConfigFlags_ViewportsEnable)
		{
			style.WindowRounding = 0.0f;
			style.Colors[ImGuiCol_WindowBg].w = 1.0f;
		}

		// Setup Platform/Renderer bindings
		ImGui_ImplGlfw_InitForOpenGL(glWindow, true);
		ImGui_ImplOpenGL3_Init(glsl_version);

		for (auto window : windows)
			if (!window->Init())
				return false;

		return true;
	}

	bool Design()
```

```
	{
		// Show main window docking space
		if (!DesignMainWindowDockingSpace())
			return false;

		if (shouldClose)
			return true;

		for (Window* window : windows)
			if (window->ShouldClose()) {
				if (!window->Exit())
					return false;
				else
					windows.erase(std::find(windows.begin(), windows.end(),
window));
			}
			else if (!window->Design())
				return false;

		return true;
	}

	bool MainLoop() {
		// Main loop
		while (!glfwWindowShouldClose(glWindow)) {
			glfwPollEvents();
			ProcessInput(glWindow);

			// Start the Dear ImGui frame
			ImGui_ImplOpenGL3_NewFrame();
			ImGui_ImplGlfw_NewFrame();
			ImGui::NewFrame();

			if (!Design())
				return false;

			if (shouldClose)
				return true;

			// Rendering
			ImGui::Render();
			int display_w, display_h;
			glfwGetFramebufferSize(glWindow, &display_w, &display_h);
			glViewport(0, 0, display_w, display_h);
			glClear(GL_COLOR_BUFFER_BIT);
			ImGui_ImplOpenGL3_RenderDrawData(ImGui::GetDrawData());

			if (io->ConfigFlags & ImGuiConfigFlags_ViewportsEnable) {
				GLFWwindow* backup_current_context = glfwGetCurrentContext();
				ImGui::UpdatePlatformWindows();
				ImGui::RenderPlatformWindowsDefault();
				glfwMakeContextCurrent(backup_current_context);
			}

			glfwSwapBuffers(glWindow);

			if (!Command::ExecuteAll())
				return false;

			Time::UpdateFrame();
			//std::cout << "FPS: " << Time::GetFrameRate() << std::endl;
		}

		return true;
```

```
	}

	bool OnExit()
	{
		for (Window* window : windows)
			if (!window->Exit())
				return false;

		// Cleanup
		ImGui_ImplOpenGL3_Shutdown();
		ImGui_ImplGlfw_Shutdown();
		ImGui::DestroyContext();

		glfwDestroyWindow(glWindow);
		glfwTerminate();

		return true;
	}
};



// Render pipeline:
// Compute white x or y limits for each line
// Project lines to camera z=0 plane
// Apply white limits to shaders of lines
class Renderer {
	GLuint VAOLeft, VAORight, VBOLeft, VBORight, ShaderLeft, ShaderRight;


	static void glfw_error_callback(int error, const char* description)
	{
		fprintf(stderr, "Glfw Error %d: %s\n", error, description);
	}

	bool InitGL()
	{
		// Setup window
		glfwSetErrorCallback(glfw_error_callback);
		if (!glfwInit())
			return 1;

		// Decide GL+GLSL versions
#if __APPLE__
	// GL 3.2 + GLSL 150
		const char* glsl_version = "#version 150";
		glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
		glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 2);
		glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);  // 3.2+ only
		glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);            // Required on Mac
#else
	// GL 3.0 + GLSL 130
		glsl_version = "#version 130";
		glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
		glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 0);
		//glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);  // 3.2+ only
		//glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);            // 3.0+ only
#endif

	// Create window with graphics context
		glWindow = glfwCreateWindow(1280, 720, "StereoOriginal", NULL, NULL);
		if (glWindow == NULL)
			return false;
		glfwMakeContextCurrent(glWindow);
```

```
		glfwSwapInterval(1); // Enable vsync

		// Initialize OpenGL loader
#if defined(IMGUI_IMPL_OPENGL_LOADER_GL3W)
		bool err = gl3wInit() != 0;
#elif defined(IMGUI_IMPL_OPENGL_LOADER_GLEW)
		bool err = glewInit() != GLEW_OK;
#elif defined(IMGUI_IMPL_OPENGL_LOADER_GLAD)
		bool err = gladLoadGL() == 0;
#else
		bool err = false; // If you use IMGUI_IMPL_OPENGL_LOADER_CUSTOM, your loader is
likely to requires some form of initialization.
#endif
		if (err)
		{
			fprintf(stderr, "Failed to initialize OpenGL loader!\n");
			return false;
		}

		return true;
	}

	void CreateShaders()
	{
		std::string vertexShaderSource1		= GLLoader::ReadShader("shaders/.vert");
		std::string fragmentShaderSourceLeft1 = GLLoader::ReadShader("shaders/Left.frag");
		std::string fragmentShaderSourceRight1 =
GLLoader::ReadShader("shaders/Right.frag");

		const char* vertexShaderSource = vertexShaderSource1.c_str();
		const char* fragmentShaderSourceLeft = fragmentShaderSourceLeft1.c_str();
		const char* fragmentShaderSourceRight = fragmentShaderSourceRight1.c_str();

		ShaderLeft = GLLoader::CreateShaderProgram(vertexShaderSource,
fragmentShaderSourceLeft);
		ShaderRight = GLLoader::CreateShaderProgram(vertexShaderSource,
fragmentShaderSourceRight);
	}
	void CreateBuffers()
	{
		glGenVertexArrays(1, &VAOLeft);
		glGenBuffers(1, &VBOLeft);
		glGenVertexArrays(1, &VAORight);
		glGenBuffers(1, &VBORight);
	}

	void DrawLineLeft(const Pair& line)
	{
		glBindBuffer(GL_ARRAY_BUFFER, VBOLeft);
		glBufferData(GL_ARRAY_BUFFER, sizeof(Pair::p1) * 2, &line, GL_STREAM_DRAW);

		glVertexAttribPointer(GL_POINTS, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float),
(void*)0);
		glEnableVertexAttribArray(GL_POINTS);
		glBindBuffer(GL_ARRAY_BUFFER, 0);


		// Apply shader
		glUseProgram(ShaderLeft);

		glBindVertexArray(VAOLeft);
		glDrawArrays(GL_LINES, 0, 2);
	}
	void DrawLineRight(const Pair& line)
```

```
	{
		glBindBuffer(GL_ARRAY_BUFFER, VBORight);
		glBufferData(GL_ARRAY_BUFFER, sizeof(Pair::p1) * 2, &line, GL_STREAM_DRAW);

		glVertexAttribPointer(GL_POINTS, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float),
(void*)0);
		glEnableVertexAttribArray(GL_POINTS);
		glBindBuffer(GL_ARRAY_BUFFER, 0);


		// Apply shader
		glUseProgram(ShaderRight);

		glBindVertexArray(VAORight);
		glDrawArrays(GL_LINES, 0, 2);
	}


	void DrawSquare(WhiteSquare& square)
	{
		// left top
		glBindBuffer(GL_ARRAY_BUFFER, square.VBOLeftTop);
		glBufferData(GL_ARRAY_BUFFER, WhiteSquare::VerticesSize, square.leftTop,
GL_STREAM_DRAW);

		glVertexAttribPointer(GL_POINTS, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float),
(void*)0);
		glEnableVertexAttribArray(GL_POINTS);
		glBindBuffer(GL_ARRAY_BUFFER, 0);

		// Apply shader
		glUseProgram(square.ShaderProgramLeftTop);
		glBindVertexArray(square.VAOLeftTop);
		glDrawArrays(GL_TRIANGLES, 0, 3);



		// right bottom
		glBindBuffer(GL_ARRAY_BUFFER, square.VBORightBottom);
		glBufferData(GL_ARRAY_BUFFER, WhiteSquare::VerticesSize, square.rightBottom,
GL_STREAM_DRAW);

		glVertexAttribPointer(GL_POINTS, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float),
(void*)0);
		glEnableVertexAttribArray(GL_POINTS);
		glBindBuffer(GL_ARRAY_BUFFER, 0);

		// Apply shader
		glUseProgram(square.ShaderProgramRightBottom);
		glBindVertexArray(square.VAORightBottom);
		glDrawArrays(GL_TRIANGLES, 0, 3);

	}

	void DrawObject(StereoCamera* camera, SceneObject* o) {
		for (auto l : o->GetLines()) {
			// Lines have to be rendered left - right - left - right...
			// This is due to the bug of messing shaders.
			glStencilMask(0x1);
			glStencilFunc(GL_ALWAYS, 0x1, 0xFF);
			glStencilOp(GL_KEEP, GL_REPLACE, GL_REPLACE);
			DrawLineLeft(camera->GetLeft(l));

			glStencilMask(0x2);
```

```
                        glStencilFunc(GL_ALWAYS, 0x2, 0xFF);
                        glStencilOp(GL_KEEP, GL_REPLACE, GL_REPLACE);
                        DrawLineRight(camera->GetRight(l));
                }
        }

public:
        GLFWwindow* glWindow;


        const char* glsl_version;
        float LineThickness = 1;
        glm::vec4 backgroundColor = glm::vec4(0, 0, 0, 1);

        WhiteSquare whiteSquare;

        void Pipeline(const Scene& scene)
        {
                glDisable(GL_DEPTH_TEST);
                glClearColor(backgroundColor.r, backgroundColor.g, backgroundColor.b,
backgroundColor.a);

                // This is required before clearing Stencil buffer.
                // Don't know why though.
                // ~ is bitwise negation
                glStencilMask(~0);

                glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);
                glLineWidth(LineThickness);

                glEnable(GL_STENCIL_TEST);

                for (auto o : scene.objects)
                        DrawObject(scene.camera, o);

                DrawObject(scene.camera, scene.cross);

                glStencilMask(0x00);
                glStencilFunc(GL_EQUAL, 0x1 | 0x2, 0xFF);
                glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);

                DrawSquare(whiteSquare);

                glDisable(GL_STENCIL_TEST);
                glEnable(GL_DEPTH_TEST);
        }


        bool Init()
        {
                if (!InitGL()
                        || !whiteSquare.Init())
                        return false;

                CreateShaders();
                CreateBuffers();

                return true;
        }
};
```

Додаток В

Реалізація процедур для побудови каркасних стерео зображень з урахуванням багаторакурсного спостереження

Текст програмного модулю

УКР.НТУУ”КПІ”ІМ.ІГОРЯ_СІКОРСЬКОГО_ТЕФ_АПЕПС_ТВ6128_20Б

Аркушів 10

Київ — 2020

АННОТАЦІЯ

Додаток містить опис головного модуля програми, що складається з точки входу програми, класу, що відповідає а графічний інтерфейс на головний цикл програми та клас, що відповідає за малювання сцени.

Модуль створений у середовищі Microsoft Visual Studio 2019 Community, з використанням мови програмування C++17. Для роботи з графікою використано бібліотеку OpenGL + GLFW. Для роботи з графічним інтерфейсом використано бібліотеку ImGui. Для роботи з векторами та кватерніонами було використано бібліотеку GLM.

ЗМІСТ

1. ЗАГАЛЬНІ ВІДОМОСТІ

Назва програми - StereoOriginal.

Для роботи програми мають бути виконані наступні вимоги:

- Операційна система Microsoft Windows 10 x64 (потенційно працює на win7 та інших сумісних системах);
- Встановлено Microsoft Visual C++ 2015 Redistributable Update 3 RC чи вище;
- Центральний або Графічний процесор з підтримкою OpenGL 3 та вище;
- Наявність підключеної камери (опціонально).

Програма написана мовами програмування C++17 та GLSL (OpenGL Shading Language).

2. ФУНКЦІОНАЛЬНЕ ПРИЗНАЧЕННЯ

Програма розв’язує задачі:

- Побудови тривимірних моделей;
    - Обмежене побудовою ламаних та їх вичавлюванням;
    - Можлива імітація кривої за допомогою моментального режиму інструмента малювання;
- Трансформації тривимірних моделей у локальних та світових координатах;
- Рендеринг сцени у стереоскопічному режимі;
- Визначення позиції користувача перед монітором;
    - Користувач має знаходитися на відстані від 20 до 70 сантиметрів від камери, та знаходитися у полі зору камери для функціонування даного функціоналу. Також погане освітлення негативно впливає на точність роботи функціоналу;
- Багаторакурсного спостереження;
    - Ракурс спостереження є не постійним через похибку обчислення позиції користувача, тож зображення може хитатися.

3. ОПИС ЛОГІЧНОЇ СТРУКТУРИ

Логічна структура програми складається з набору логічних модулів, що досить тісно пов’язані між собою. Модулі у свою чергу складаються з набору класів, що можуть бути замінені, видалені, чи створені нові.

Структура модулів має наступний вигляд:

- Графічний інтерфейс програми;
    - Випадаюче меню;
    - Загальні вікна програми;
    - Рендеринг сцени;
- Функціональний модуль;
    - Інструментальні вікна програми;
    - Інструменти;
- Доменний модуль;
    - Сцена;
    - Об’єкти сцени;
- Файловий менеджер;
    - Бінарні читання\запис;
    - Json читання\запис;
- Інфраструктурний:
    - Робота з часом;
    - Логування;
    - Завантаження шейдерів;
    - Події;
    - Команди;
    - Глобальні налаштування інструментів;
    - Розширення шаблонів;
- Визначення позиції користувача перед камерою,

де 1 рівень списку – логічна назва модулю,

2 рівень списку – функції, цей модуль забезпечує.

Слід зазначити, що модулі, описані вище не є цілісними. Вони, здебільшого складаються з набору класів, що можуть навіть не знаходитися у одному файлі. Це обумовлено специфікою роботи мовою програмування C++. Об’єкти мають бути об’явлені до їх використання, тож часто базові класи розташовані в окремих файлах, які можуть бути використані до імплементації конкретних класів.

Графічний модуль є водночас і головним модулем, який містить у собі головний цикл програми. Він пов’язаний з вікнами та інфраструктурою проекту. Вікна, у свою чергу, посилаються на функціональний модуль для відображення інструментів у вікні програми.

Функціональний модуль відповідає за усі операції над об’єктами, що виконуються користувачем окрім роботи з файловою системою. Посилається на Доменний та Інфраструктурний модулі.

Доменний модуль посилається лише на інфраструктурний модуль для отримання інформації про глобальні налаштування інструментів, такі як режим координат, дія при зміні батька тощо.

Файловий менеджер посилається на доменний модуль для читання та запису об’єктів сцени у файл.

Інфраструктурний модуль не посилається на жодні модулі проекту, так само як і модуль визначення позиції користувача перед камерою.

4. ВИКОРИСТОВУВАНІ ТЕХНІЧНІ ЗАСОБИ

Для роботи програми мають бути виконані наступні вимоги:

- Комп’ютер під управлянням операційної системи Microsoft Windows 10 x64 (потенційно працює на win7 та інших сумісних системах);
- Центральний або Графічний процесор з підтримкою OpenGL 3 та вище;
- Наявність підключеної камери (опціонально).

5. ВИКЛИК І ЗАВАНТАЖЕННЯ

Програма встановлення не потребує. Перемістити набір файлів, що потрібні для функціонування програми у обрану папку та запустити виконуваний файл проограми. Відклик програми залежить від потужості центрального процесора комп’ютера та складності сцени, що проектується. Велика кількість відрізків знижує продуктивність систему в цілому і повільно реагувати на дії користувача.

6. ВХІДНІ ДАНІ

Вхідними даними до програми є взаємодія користувача з графічним інтерфейсом програми, положення користувача перед камерою (за увімкненої опції), файл конфігурації графічного інтерфейсу програми, файли шейдерів для рендерингу, набір натренованих моделей для розпізнавання позиції користувача та набір збережених сцен користувача у бінарному або json вигляді.

7. ВИХІДНІ ДАНІ

Вихідними даними є зміни в графічному інтерфейсі програми, зокрема відмальовка сцени, та файл конфігурації графічного інтерфейсу програми, що змінюється коли користувач переміщує вікна чи змінює їх властивості. А також набір збережених сцен користувача у бінарному або json вигляді.